Contents

Chapter 1

Instead of Preface

What this book covers

This book covers the most important Vim concepts. Besides Vim, it also covers some of the best learning strategies known to me (the author), and which are proven to work.

Having an understanding of how our brain works, and being aware of simple but effective learning techniques, will help you to learn Vim (and any other topic) faster than ever before.

What you need for this book

You need to have a strong enough motivation and wish to learn Vim. That's necessary. But it's not enough. Motivation is not the factor which will change your life. If someone is a fool and becomes motivated, he merely becomes a motivated fool. That won't lead to a positive change.

You'd have to be disciplined in going through the challenge of improving your Vim skills. This way you'll learn Vim quickly. But learning quickly is not equal to learning easily. You'll have to put in some serious work.

You might get stuck at some point. You might even think of giving up. You might think *Vim is too hard*. I'll tell you one thing: **Don't wish Vim was easier, wish you were better!**

Who this book is for

This book is for everyone who'd like to learn Vim. Whether you're a beginner and you're starting from scratch, or a more advanced user, I believe that you'll find this book very useful.

Vim versions

All commands, shortcuts and examples in this book are tested using Vim 8. You're free to use different versions of Vim while learning (like gvim, macvim, neovim), but some shortcuts and commands might behave differently than those presented in the book.

That's why I recommend you simply use Vim 8 while learning from this book. Once you're done with it, you can think of moving to your preferred version of Vim.

Conventions

Before we start with the real stuff, pay attention to the typographical conventions that are used in this book.

Key presses

Esc - Indicates that the named key (in this case key *Esc*) should be pressed.

Ctrl-p - Keys separated by `-` should be pressed simultaneously. In this example, *Ctrl* key should be held down while the *p* key is pressed.

Normal mode commands

These are commands you type while you're in Normal mode. For example, command to delete four words is the command: `d4w`. The named characters should be entered in order.

Command line commands

You'll learn more about commands later. Now just pay attention to the format of this commands.

Example of command line command (enable line numbers):

```
:set nu
```

This command enables line numbers.

All commands starting with `:` should be executed in Normal mode. If you're in a different mode, press *Esc* before you type a command. Within the Normal mode you can run Ex commands. Normal mode is also known as command mode.

Fonts

A code block will be used to show code samples, multiple commands and some examples. Here's an example of presenting the *read* command and its shorter version:

```
:read
:r
```

A `fixed width font` is used for filenames, code, variable names and commands.

Other formatting

There are commands shown in this format: `:ju[mps]`. This means that you can execute this command in two ways:

```
:jumps
:ju
```

As you can see, everything between `[` and `]` is optional.

Very often I'll use bold characters in words explaining commands. For example: The command to **d**elete is `d`.

Bold characters in cases like this will indicate the command abbreviation, which should help you remember Vim commands easier.

Commands are case sensitive. If you see `g` in command, that's what you type. If you see `G`, it means you type capital G.

If some commands should be typed in a system command line, they will be shown in a block like this:

```
$ vim
```

This is how you start Vim from command line. Characters `$` are used to indicate the command line. You should type everything after that - in this case `vim`.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem. I do take the protection of my copyright and licenses very seriously. If you come across any illegal copies of my works in any form on the Internet, please email me at contact@jovicailic.org with a link to the suspected pirated material.

I will greatly appreciate your help in protecting my work and ability to bring *you* more valuable content.

Mastering Vim Quickly Newsletter

I recommend you subscribe to my weekly newsletter *Mastering Vim Quickly Newsletter* available at http://masteringvim.com.

Every week I'll send you the best tips, tricks and news on Vim. Some of them will be from this book, but also lots of them are not covered in the book. Simply, there's too many things you can learn about Vim. That's why getting a tiny bit of knowledge once a week is a great way to steadily improve your Vim skills.

Why this name

First time I saw an advanced Vim user working on his code, my impression was like "wtf just happened". He would edit his code in a way I've never seen before. Then, when I realized how easy it is to do this, I thought "omg I have to try this!".

That's how I came up with *Mastering Vim Quickly: From WTF to OMG in no time*.

Questions

If you have a problem with any aspect of this book, feel free to contact me at any time. My email is: contact@jovicailic.org. I'll do my best to address the problem.

Chapter 2

Introduction

There's so much you want to do in life, and so little time. It's the story of our modern lives. Now, you want to learn Vim. Congratulations for making this decision! Now think, what's holding you back from getting started? It's hard? It takes time? Effort?

Over the years, I have learned two uncomfortable truths related to learning. First: skills take time and effort to master. And second: many things aren't fun until you're good at them.

While learning any skill, there is a period of time in which you're horribly unskilled, and you're painfully aware of that fact. The same applies when learning Vim. This book presents my personal quest to quickly learn Vim, and as such, it will help you to acquire new Vim skills in record time.

One of the beautiful things about learning any subject is the fact that you don't need to know everything. What's important is that you only need to understand a few critically important concepts that provide most of the value. The same goes for Vim.

Mastering Vim Quickly is a set of foundational Vim concepts you can use to get things done. Once you master the fundamentals, you can accomplish even the most challenging Vim magic with surprising ease. And you will love it!

Over the past few years, I read several books on Vim, passed through hundreds of tutorials and tips, used Vim from a few to over 10 hours a day and coded in more than a few different programming languages. Along the way, I've collected, distilled and refined my findings into the concepts and best tips which are presented in this book.

If you invest the time and energy necessary to learn these concepts, you'll easily be in the top 5% of the human population when it comes to productivity in coding, programming and text editing.

Think of this book as a filter. Instead of trying to absorb all of the Vim knowledge—and there's really a lot out there—use this book to help you with what matters most. This way you can focus on what's actually important: getting stuff done.

Bram Moolenaar, the creator of Vim, wrote:

> "Learning to drive a car takes effort. Is that a reason to keep driving your bicycle? No, you realize you need to invest time to learn a skill. Text editing isn't different. You need to learn new commands and turn them into a habit."

And I completely agree. This book will help you improve your Vim skills.

The art of learning

I'm a learning addict. I usually read few books every month. This is good, because I learn a lot. Then I try out what I learned, and adopt what works for me. In this short chapter, I will present the three most important principles which work for me when it comes to learning.

Don't skip them. I believe it's very important that each one of you has the same basic start when it comes to learning techniques. I really want to help you learn Vim quickly. Therefore pay attention to these principles. Once they are understood, you can use them for learning and improving on any topic, not just Vim.

Pareto principle

Italian economist and sociologist Vilfredo Pareto (1848-1923) observed that 80% of the land in Italy was owned by 20% of the population. While investigating other countries, he found the same unequal distribution of income and wealth in each.

Pareto's Principle basically states that roughly 20% of all our actions produce 80% of our results. This means that 80% of effects come from 20% of causes. Because of these numbers, it's also called 80/20 principle.

We know that it doesn't have to be 80/20. It can also be 90/10, or 70/30. That's not important right now. What you must know is that the 80/20 principle works, regardless of whether you're conscious of it working or not. This is true for your business, personal life, and everything you learn. Including Vim.

This means that, more or less, around 80% of what you've done today, has been pretty much worthless to your bottom line. You probably know there are things you should be doing, that you're just not doing for whatever reason. Maybe you're overweight or out of shape and you know you should work out more. But, back to the topic.

Why am I telling you this at all, in a book about Vim? Well, I firmly believe that this principle is true. I use it in almost every important area of my life, and it gives me very good results. I also used this principle to quickly master Vim. This book provides you 20% of the most important Vim fundamentals, which will help you to learn Vim really fast.

Mini habits

In order to learn Vim, you need to commit to it.

Think of the last time you made a commitment to learn something. Or to change something in your life. It was easy to make a commitment, wasn't it? Maybe you even had a plan! Then, fast forward a few weeks. Where's that commitment? Gone, right? You're not so motivated anymore. Your willpower is close to zero.

I know this story very well. I've been there, and done that. I had to read a lot and try out what works to find my way out. And I finally did! In the next couple of paragraphs, you'll read the summary. That's all you need to know.

When you commit to something, the best way to reach your goal is to create a habit. In this case, your goal would be mastering Vim quickly, and the habit you need to adopt is to regularly learn.

The biggest barrier to forming new habits is usually the fact that it takes discipline to keep doing something you don't really feel like doing. I found a workaround for this. It's called the mini habit. This is a game changer!

Here's my own example: As I was working on this book, sometimes I struggled to start writing. But once I start, I can easily write for longer periods of time. The problem is that, on some days, I don't want to start writing. So I won't. I'm pretty sure you've experienced something similar, even with a different activity.

The key to forming a habit is the consistency. So I decided and promised myself that, no matter what, I'll stick to writing every day for the next month. Here's what happened: for the first couple of days I was motivated to stick to the plan. Then my motivation got lower (always does), so I used my willpower to keep going.

However, after a week or so, I missed a day. And once I missed one day, it's was pretty easy to miss another. So my plan failed.

That's the biggest problem with forming new habits. If you're not consistent, you can't create a habit. So I finally understood the problem. When you feel resistance to something, you probably won't do it. If you don't feel like starting some activity (whether it's writing, running or anything else), motivation and willpower can't really help for the long term.

With mini habits, you make a workaround for this resistance. And when there's no resistance, you just start and do what you should.

The workaround in my example looked like this: I've made a commitment to write just 50 words a day. One paragraph or one Vim tip. That was my daily goal. It was very easy to achieve. I didn't feel resistance to write only 50 words. Even if I was having a bad day, I was still able to find a couple of minutes and write those 50 words! Anyone can do that. Actually, I realized that it was easier to write those 50 words, than it is was to not write them.

Why? Because if you make a commitment, and if, no matter how good or bad you feel, you can't

force yourself to write the damn 50 words, then you're basically saying to yourself that you're a big loser. Your pride won't let you fail at something so ridiculously small. Especially if it takes less than two minutes to complete.

This was a game changer for me.

You might think: "Yeah, but there's nothing you can achieve with writing 50 words per day . . . " Well, that's absolutely wrong. How so? You can try it. Sit down to write 50 words. Or trust me. When I would start with writing (with the 50 written words goal in mind), I would usually write far more than 50 words. Because once I would start, it was difficult to stop.

And this is the real power I want to share. No matter how you feel. No matter how busy you are. It's very hard to fail with such a tiny commitment. You can use this strategy for any other activity as well, it really works great.

The whole trick for me was to make a commitment to write just 50 words every day. Your commitment could be learning Vim quickly. That's why you must always go in with the intention to complete the smallest possible step. If you make it bigger, you will start to feel resistance.

1% improvement per day

You can't master any skill in one day. You have to improve a little every singe day. It compounds. That's how you should approach learning Vim as well.

Every day matters. You either increase your skill level for 1% or decrease it. It's your choice. In the beginning, there's no real difference between making a choice that is 1% better or 1% worse. It won't impact you today. But, over time these small choices compound.

When 1% compounds every day, it doubles every 72 days. If you commit to improve your Vim skills 1% every day, in one year your skills will be 38 times better!

How do you know how much is 1%? Well, when it's about Vim, it's hard to measure it. I would suggest that you decide what your 1% is going to be. It can be reading one page of this book. Or learning one new Vim feature, command or trick.

Another way is to dedicate a fixed time for learning Vim every day. Let's say that you dedicate 20 minutes every day to learning Vim. During those 20 minutes, you'll improve your Vim skills—sometimes by 1%, sometimes by less or more. It's not important to be that precise. What's most important here is consistency. Keep improving your skills every single day.

No Experience Necessary

Don't worry if you're a complete beginner. I don't assume that you're already good in Vim (but this book will still be very useful if you are!). You'll find the information in this book more valuable and practical than anything you learned from other Vim resources.

Each chapter is packed with examples that support detailed explanations of all the important concepts, and they are presented in a way that helps you avoid the confusion that I faced when I was learning. With this book and plenty of practice, you will be amazed at how quickly you can go from complete beginner to super productive pro.

Mastering Vim Quickly is for anyone who wants to learn Vim, but who possibly doesn't know where to start, or has tried to learn but struggled to make progress, or is intimidated by how difficult Vim appears to be. This book is designed to give you the head start I didn't have. Wherever you are, if you want to learn Vim, this book will help you learn smarter. Start exploring Vim today, and get to productivity fast!

Chapter 3

Mastering Vim - Basics

Installing Vim 8

If you don't have Vim already installed in your OS, you should do it now. I'll assume that you're using some sort of Linux. However, even if you use Vim on PC or Mac operating systems, it's okay.

In order to install Vim, you should run the following command:

Gentoo: `emerge vim`

Ubuntu/Debian: `apt-get install vim`

CentOS/Fedora: `yum install vim`

MacOS: `brew install vim`

Now when you have installed Vim, we can start. Vim is also supported on Windows operating systems, but be aware that some commands won't work as presented here. However, most of what we cover will work.

Vim philosophy

Vim is different from other text editors. Although it looks hard to learn, there are only two main ideas you need to really understand it. Those two are actually what makes Vim so different from other text editors. They are **modal editing** and **operators**.

Modal editing

The purpose of Vim is to enable you to edit text **effectively**.

When you're writing code, how do you spend most of your time in the editor? You're likely not typing most of the time, but rather moving around through existing code and editing here and there.

Because you spend more time editing than entering the text, Vim makes editing and navigation more powerful than any other text editor.

For example, you want to jump to the end of your file? Simply press `G`. You want to jump to the top of your file? Hit `gg`.

You might wonder, how Vim will know whether you want to type letter `G` as a part of some word, or you want to jump to the end of your file. That's why Vim has modes. What happens when you press `G` depends on the mode.

Pressing `G` in Normal mode moves the cursor to the last line in your file. Pressing the same key in Insert mode will simply add `G` to your text. When you start Vim, it will be in Normal mode by default. If you want to enter text, you have to enter Insert mode.

You can do it by pressing `i` while in Normal mode. When you're done with typing your text, press *Esc* and you'll return to Normal mode.

Besides Normal and Insert modes, there are a few other important modes.

Operators

If you are used to moving your cursor using arrow keys, "page up," "page down," "home" and "end," you'll be very amazed what Vim can offer. You'll be able to move your cursor faster and with better precision.

For example, you want to delete a text from the middle of the sentence to the end of line? Place the cursor to the position from where you want to delete, and press `d$`. Here `d` stands for **d**elete, while `$` means end of the line.

It may look strange now, but you'll see it's very easy. In fact, there's a whole Vim language which you'll learn later. And once you do, you'll love it!

In the previous example, `d` is the operator command. Operator commands (or operators) can delete, change or insert text, copy or format it, etc. You'll learn more about commands soon.

Starting Vim

When you want to start Vim from the command line, you have many options you can provide to it. Here's the list of some options which you'll find useful sooner or later:

- `+NUM` - The cursor will be positioned on the line "NUM" for the first file you open.
- `+/{pattern}` - The cursor will be positioned on the first line containing "pattern" in the first file you open.
- `+cmd` or `-c cmd` - Command "cmd" will be executed after the first file has been read. It's interpreted as an Ex command. You'll learn about Ex commands soon.
- `-x` - Use encryption to read or write files. You need to use this option only the first time for a given file. Every following time you'll be asked for the password even without this option. The encryption implementation is not strong, so don't rely only on this protection for your important data.

For example, to open file `my_file.txt` with the cursor positioned on line 33, you would start Vim like this:

```
$ vim +33 my_file.txt
```

This can be very handy when you're debugging your code, and you know at which line you have an error to fix. Or, another case when it can be very useful is this SSH problem:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:+lqGiMAmByST3y6wER8uv5INGlIagx2pOVIMdAPc6LSRKI.
Please contact your system administrator.
Add correct host key in /home/jole/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/jole/.ssh/known_hosts:466
ECDSA host key for secret.masteringvim.com has changed and you have requested strict checking.
Host key verification failed.

jole:~/ $ vim +466 .ssh/known_hosts
```

This SSH issue can be resolved by removing the line `466` from `~/.ssh/known_hosts` file. The solution listed above is a nice way to do it. However, the quickest way to do this is to run the following command in your terminal: `vim .ssh/known_hosts +"466d|x"`

Chapter 4

Your first Vim session

To run Vim, open your terminal, and type: `vim`

This will start Vim, as always, in Normal mode. To start with inserting some text, you need to switch to Insert mode. You do this by simply pressing `i`. Feel free to type some text, and then try moving your cursor around with arrow keys, so you can gain familiarity.

Now, when you're done with typing, let's save this text in a file. To do so, you need to get back to Normal mode. You do this always by pressing the *Esc* key.

If you have never used Vim before, you're probably a bit confused. What in the world is a mode? I'll explain it to you very soon. For now, just try to understand that Vim is different than any other text editor you've used. The concept of modes is very simple: you want to insert text—then enter the Insert mode first. You're done with typing? Exit Insert mode.

Now, whenever you want to type a command, you need to type `:`. This is how every command starts. The command to **save the file and exit Vim** is `:wq <filename>`. Letter `w` comes from write, letter `q` comes from quit. So, type `:wq test.txt` and press *Enter*. Voilà, you've just created a file called `test.txt`.

If you want to open a file you've just created, just type in your terminal: `vim test.txt`. Press key `i` and add some more text to your file. But, you don't want to save it this time. In order to close the file without saving it, and exit Vim, you'll need to get back to Normal mode (press *Esc*) and type `:q!`.

Simple as that. Of course, there are several more ways to open and close files in Vim, but this is enough for your fist Vim session.

Now, let's do something cool. You won't understand everything now, but you'll have a complete understanding of this example when you read the entire book.

If you use https://github.com and `git` in the command line, you've probably had to squash some of your commits at times. This example will show you how quickly you can do it with Vim, even

though you're just at the very start.

In case you don't know what I'm talking about, never mind—just copy the snippet below to Vim and try out the example.

Info: You're squashing your commits, here's the list of them:

```
pick e08be68 Add initial Rexfile
pick 5b4143f Fixes
pick 855be75 fix
pick d7a5285 Initial work
pick 59e82a2 add more stuff
pick 34cfc9c Fixes
```

Task: Change word `pick` from line 2 to last line (line 6) to `s`.

Solution:

1. Open Vim. Press `i` to enter Insert mode. Then copy the text from above. Paste the text in Vim, using the shortcut for pasting in your terminal.
2. Now press *Esc* to get back to Normal mode. Use arrows to place your cursor on the beginning of the second line.
3. Press *Ctrl-v*. Then press *right-arrow* three times. You'll notice that you just selected word `pick` in second line.
4. Now press *down-arrow* four times. You'll notice that you've selected all words `pick` starting from second to last line.
5. While this block of words is selected, press `c`. That's a command for **c**hanging text.
6. All words in your selection will be deleted, and cursor placed on second line. Mode will change to Insert. Now just type `s`. That's what we want to replace word `pick` with.
7. Last final step - press *Esc*. And voila! You did it!

Here's how the result should look like:

```
pick e08be68 Add initial Rexfile
s 5b4143f Fixes
s 855be75 fix
s d7a5285 Initial work
s 59e82a2 add more stuff
s 34cfc9c Fixes
```

This might look complicated, but trust me, it's very easy. Keep reading, and soon you'll understand what actually happened in this example.

Chapter 5

Vim Concepts

Vim is a different text editor from all the other editors. That's why you need to understand some Vim concepts, as you probably have never experienced concepts like these with different text editors.

Modes

If you want to understand and learn Vim, you need to understand Vim modes. Vim has twelve modes in total. However, in your daily use, you'll need to use 4 or 5 of them regularly. It's very important to understand those.

Here are the most important ones:

Normal mode - When you start Vim, by default you'll be in Normal mode. In this mode you can enter all the normal editor commands. It's mostly used for navigation and text manipulation. Advanced Vim users spend most of their time in Normal mode. A good habit to adopt and keep in mind: whenever you're not typing, it's better to get back to Normal mode.

Insert mode - As the name says, this is for inserting new text. In this mode you can also run some of the commands. By default, "– INSERT –" is shown at the bottom of Vim window.

Command mode - In this mode you can run Ex commands (like `:set number`), enter search patterns (like `/word`) and enter filter commands. After running the command, Vim returns to Normal mode.

Visual mode - For navigation and manipulation of text selections. It's similar to Normal mode, but the movement commands extend a highlighted area. When you use a non-movement command, it's executed for the highlighted area. By default, there is "– VISUAL –" shown at the bottom of the window.

Insert Normal mode - When you're in Insert mode, and press *Ctrl-o*, you'll enter this mode. It's similar to Vim Normal mode, but after executing one command, Vim returns to Insert mode. By default, '– (insert) –' is shown at the bottom of the window.

Commands

Commands are probably the most important concept in Vim. Most of what you'll do within Vim will be the result of executing different kinds of commands. In general, we can say that there are three ways of executing commands in Vim:

Ex commands

This is any command you can run as `:{command}`, for example `:help`. You will use them often. You can see the entire list of these commands (it's very long, don't do it now) by running `:help ex-cmd-index`.

Mapped commands

Any (more complex) commands, which we map or bind to some keys for easier access, belong to this group. You'll usually add these commands to your `.vimrc` file. You'll learn more about these in chapter *Mappings*.

Editing commands

These are the commands which you'll usually use in Normal and Insert mode. These are commands like `d4w` which will delete four words after your cursor. You'll learn many of these commands in chapter *Do you speak Vim?*.

Chapter 6

Working with files

Opening files

Here are the two most common methods for opening files in Vim.

Method 1 - Open file from terminal

Once you open your terminal, type `vim` and then the filename. For example:

```
$ vim /etc/passwd
```

Method 2 - Open file from Vim

When you start Vim by running `vim` in your terminal, there will be no files loaded, by default. Then, run command: `:e <filename>` to open a file in your existing Vim session.

For example:

```
$ vim

:e /etc/passwd
```

Very often you'd like to get the content of some other file into your current opened file. Actually, to be more precise, instead of "current opened file," from now on, we'll use the term "current buffer."

So when you open an existing file, the content of this file is loaded in one Vim buffer. You'll learn much more about buffers later, but for now, just remember that buffer is a piece of memory that's been loaded with the content of a file.

Of course, you could open the second file, copy the content you need, return to first file, and paste. But, there's a better way.

Vim has a **r**ead command:

```
:read
:r
```

You can use this command to insert a file, or the output from a system command, into the current buffer.

Here are a few examples of how you can use it:

Command	Description
`:r file.txt`	Insert the file file.txt below the cursor in current buffer.
`:0r file.txt`	Insert the file file.txt before the first line.
`:r!sed -n 2,8p file.txt`	Insert lines 2 to 8 from file file.txt below the cursor.
`:r !ls`	Insert a directory listing below the cursor.

Of course, all of these should be run in Normal mode. The last command will work only in Linux or macOS.

Related tip: Using command **gf**, you can open a file whose name (or path) is under or after the cursor. Remember it as "goto file". Similarly, using **gx** command, you can open links in your default browser.

Closing files

There are more than a few ways to close a file in Vim. Here are some of the most common ways:

Command	Description
`:wq`	Save currently opened file and exit Vim (even if file is not changed).
`:x`	Exit Vim but write only when changes have been made.
`ZZ`	Equivalent to `:x`. Notice there's no `:`. This is a key press.
`:q!`	Exit Vim without saving currently opened file.
`:qa`	Exit all open files in current Vim session.

Saving files

There are several ways to save a file in Vim. When you save the file, you actually write the contents of the buffer to the disk. That's why, the command for saving is "**w**rite".

Here are the most common commands you should know:

Command	Description
`:w`	Save currently opened file (which was previously saved).
`:w file.txt`	Save currently opened file as `file.txt`.
`:w! file.txt`	Save file as `file.txt` with overwrite option.
`:sav file.txt`	Save current buffer as a new file `file.txt`.
`:up[date] file.txt`	Like `:w` but only save when the buffer has been modified.

Navigation

In order to be truly efficient with Vim, you have to learn how to properly navigate through your files, buffers, help system, etc. This section will enable you to improve your navigation speed drastically.

Basic movement

Just like in any other text editor, you can use arrow keys (Up, Down, Left, Right) to move around within your text in Vim. But, in Vim, there's an alternative.

Most advanced Vim users prefer to keep their hands and fingers around the home row on a keyboard. This is possible, because instead of arrows, you can use keys *h*, *j*, *k* and *l* for navigation.

- *h* - left
- *j* - down
- *k* - up
- *l* - right

In the beginning, it might be hard to get used to these. The first problem is to remember which key does what. Here's my suggestion for how to remember them:

Look at your keyboard and notice how *j* looks like an arrow *down*, with a half head. So *j* takes your cursor one position down. These keys are positioned in this way (assuming you're using a qwerty keyboard layout): *H J K L*.

The key far **right** is *l* and it will take you to the **right**. The key far **left** is *h* and it will take you to the **left**. Two keys remain in the middle, *j* and *k*.

As we already said, we can easily identify key *j* as an arrow pointing down. The only key left is *k*, which must take you up—as all the other directions we already covered.

A lot of people suggest that you disable arrow keys, so you get used to **h j k l** keys faster. But I think differently. You shouldn't disable arrow keys at all. Feel free to use them as long as you feel comfortable with them. Just, from time to time, as you progress with Vim, try using *h j k l* sometimes.

Bit by bit, you'd probably realize very quickly the great advantage of *h j k l*. You won't have to move your fingers from the key home row, which is very comfortable and efficient.

But it's best not to hurry with *h j k l* keys. Trying to force yourself to use only them, and not arrow keys can often have a negative effect. So just ignore what everyone else is saying and do whatever feels more comfortable.

Navigate through words

It will take you some time to get used to, but I highly recommend that you try to adopt this kind of navigation. When you operate on a single line (or even a few), instead of moving one character up, down, left or right, you can move between words. Also, there are some other useful shortcuts you should remember:

Key	Description
`w`	Go to the start of the next **w**ord
`W`	Go to the start of the next **W**ORD
`e`	Go to the **e**nd of the current **word**
`E`	Go to the **e**nd of the current **WORD**
`b`	Go to the previous (**b**efore) **word**
`B`	Go to the previous (**b**efore) **WORD**

`WORD` consists of a sequence of non-blank characters. It's always delimited by white space. On the other hand, `word` is delimited by non-keyword characters, which are configurable. Remember that `word` ends at a non-word character, such as a `.`, `-` or `)`.

For example, in sentence:

```
Vim "navigation" is not-so difficult!
```

we have 5 `WORDS`: `Vim` `"navigation"` `is` `not-so` `difficult!`, all delimited by white space. However, we have 10 `words`.

So if you're navigating through source code, and want to stop at delimiters and characters like `( ) . { } , $` use `w`. If you're working with text and want to skip these, then use `W`. For more information, take a look at `:help 03.1`.

Note: All of these commands for navigation can take a number as a prefix. For example `3w` will take you to the start of the 3rd next word, while `6j` will take you six lines below.

Scrolling pages

When you'll be working with a large file, you need to move through the file differently. To scroll your file page by page, you can use the following shortcuts:

Shortcut	Description
`Ctrl-d`	Scroll **d**own half page
`Ctrl-u`	Scroll **u**p half page
`Ctrl-f`	Scroll down **f**ull page (or **f**orwards)
`Ctrl-b`	Scroll up full page (to **b**eginning, or **b**ackwards)

Jumping around the file

Vim offers you simple ways to go to the beginning or end of your file. This can be very handy when you're working with large files. Beside these, in the table below, there are a few more handy shortcuts for jumping through the file:

Command	Description
`gg`	Go to the top of the file
`G`	Go to the bottom of the file
`{`	Go to the beginning of current paragraph
`}`	Go to the end of current paragraph
`%`	Go to the matching pair of (), [], {}
`50%`	Go to line at the 50% of the file
`:NUM`	Go to line NUM. `:28` jumps to line 28

Navigating inside the window

Here are a few handy shortcuts you can benefit from, when it comes to moving your cursor in the current Vim window:

Key	Description
`H`	Move cursor to first (**h**ighest) line in current window.
`L`	Move cursor to the **l**owest line in current window.
`M`	Move cursor to the **m**iddle of the current window.

Navigating in Insert mode

If you want to move around and make edits in Insert mode, you shouldn't, most of the time. The proper way would be to hit *Esc* to get to Normal mode, go to the correct location, make an edit, and get back to Insert mode.

For example, you could press *Ctrl-o* `F m` to move to previous `m` character and get to Insert mode.

However, sometimes, you'd find it easier to stay in Insert mode. In these cases, using arrow keys to move around is usually not fast enough. Here's what you could do:

Shortcut	Description
Shift-Right-arrow	Go to the right, word by word
Shift-Left-arrow	Go to the left, word by word

Related tip: While in Insert mode, you can press **Ctrl-o** to get back to Normal mode and execute one command, after which you'll be automatically returned to Insert mode.

Basic search

Vim has many search related options. We're going to cover some of these in the next chapter. Right now, it's important that you understand the basic theory of how search in Vim works.

All search operations are done in Normal mode.

You can search forward by pressing `/` and then typing your search pattern. Pressing *Esc* will cancel it, while pressing *Enter* will perform the search. Once you hit *Enter*, you can press `n` to search forwards for the next occurrence, or `N` to search backwards.

Now, let's try to figure out what would be the command to find the first match:

1. First match, is usually placed "on the top" of all others.
2. We already mentioned that command to jump to the top of a file is `gg`.
3. And now we know that pressing `n` while searching will take us to next search pattern occurrence.

So, if you perform a search for a pattern, and you want to jump to the first match, you need to hit `ggn`. Yup, this way we tell Vim: "go to the top of the file and find next (actually first occurrence)".

It's the same logic for the command to take you to the last match of your search. As you might already guess, it's `GN`.

You can search backwards by pressing `?` and then typing your search pattern. Pressing `n` searches in the same direction (in this case backwards), while `N` searches in the opposite direction (in this case forwards).

Searching for the current word

Vim can search for words under your cursor. In Normal mode, place your cursor to any word.

Press `*` and Vim will search forwards for the next occurrence of that word! How cool is that!

Press `#` and Vim will search backwards for the word under your cursor.

These two commands are searching for exact words. So if you perform the search using these commands while your cursor is on word `master`, it would not find the word `mastering`.

So if you don't want exact word matching, use commands `g*` and `g#` accordingly.

Search history

Vim keeps a search history. Just type `/` or `?` and use the arrow up or down keys to go through previous search commands. Of course, you can edit a command (or only a pattern) you find in history and press *Enter* to search again.

Let's say the cursor is on a word, and you want to search for a similar word. Instead of typing the entire word, here's what you can do:

1. Press `/`
2. Then press *Ctrl-r* and then *Ctrl-w*.

This will copy the current word under cursor to the command line, ready for searching. Now you can edit it and press *Enter*.

Once you're done with searching, you can hit *Ctrl-o* to jump back to your previous position (or *Ctrl-i* which will jump forwards).

What if you search for the last searched pattern again? There's no need to type the pattern again, or ever go through history. Just press `/` and hit *Enter* - an empty search pattern will repeat the last search. This will also work for `:s` and `:g` commands, which we'll cover later.

Vim also allows you to enter a count before a search. For example, what if you want to jump to the fifth occurrence of the pattern? Simply type `5/pattern`. Also, typing `6*` will search for the sixth occurrence of the current word under the cursor.

We have just covered the most important search basics.

File Manager (netrw) in Vim

Vim comes with a built-in **netrw** plugin which is a great way to browse files and directories within a Vim session. This file manager supports four ways of displaying files and directories.

You can launch netrw in several ways like:

- `:Ex` - open current directory in current Vim window (remember it as a shortcut of **Ex**plore).
- `:Ex <dir>` - open specified directory `<dir>`.
- `:Sex` - open current directory in horizontal split window (fun fact: Vim is the only editor in the world which has Sex as a command!).
- `:Vex` - open current directory in vertical split window.
- `:Tex` - open current directory in a new tab.
- `:Lexplore` - open current directory in vertical split on the left. Default setting opens files in the window to the right of the netrw window.

Try out these commands and see which one works the best for you. Personally, I prefer to have a file explorer in a vertical split, so I would usually run:

```
:40vs +Ex
```

to open current directory in vertical split window with width of 40 columns.

After you read the chapter on mapping, you'll know how to create a shortcut for this command, so you can open and close file explorer quickly.

You can change the directory listing view to show more or less information, change the sorting order or hide some kinds of files. Once you start netrw, try to hit `i` to cycle through the view types. There are four of them: thin, long, wide and tree. Once you choose your favorite, set it to be the default one in your `.vimrc` file, like:

```
let g:netrw_liststyle = 3
```

Changing how files are opened

With Vim, not only can you open files, but you can also open directories! Yes, go ahead and try to open some directory. For example, this command:

```
$ vim /home/jole
```

will open my home directory. What I'll get is a list of all files and list of all subdirectories in the directory I've opened.

When you open a directory with Vim, you actually started netrw. So yes, that's the way to start it out of Vim. Now, it's important to know that you can perform some of the basic file manager operations using netrw:

- `<Enter>` - opens the file under the cursor, or enters the directory under the cursor
- `D` - deletes the file under the cursor. You can visually select multiple files and use this command to delete all of them.
- `R` - renames the file under the cursor.
- `X` - executes the file under the cursor.
- `%` - creates a new file in the current directory. Vim will ask you for a file name and open a buffer.

By default, when you hit *Enter* to open a file, it will be opened in the same window as the netrw. That's not really practical. You would usually like to keep netrw in a side split, and load your files in another split. Fortunately, this behavior can be changed with `netrw_browse_split` option.

To make the selection permanent add the following to your `.vimrc`:

```
let g:netrw_browse_split = 4
```

Option `4` is the one I personally prefer. It open files in previous window (the current split you have beside netrw split).

Set netrw split width

How file explorer will position a window for the new file you open, can be set with the `netrw_browse_split` option. If you'd like to set the width of netrw split to 20% of your entire Vim window, put this in your `.vimrc`:

```
let g:netrw_winsize = 20
```

Editing files via SSH

One of the lesser known features of Vim is the ability to edit files remotely, over the network. This feature comes with the netrw plugin. To achieve this, netrw uses the SSH protocol, and manages remote files via the `scp` command.

Here's how to do it:

```
vim scp://user@myserver[:port]//path/to/file.txt
```

Note the double `/` for the directory on the remote host, which is needed to correctly resolve the absolute path. `[:port]` is optional.

So with the command above you can open a file located on a remote host for editing.

What actually happens in the background is that Vim uses `scp` to download the requested file from a remote machine to a local `/tmp` directory, and then opens it for editing. When you save your changes to the file, the changes are first applied to a local copy in `/tmp` directory. After that, the file is uploaded via `scp` to the remote host.

If you open a directory on a remote host, you could also use netrw to browse through remote files and directories. The important thing is to always specify the directory path with `/` at the end.

Of course, it's recommended that you use SSH keys for authentication. Otherwise, you might be asked for the SSH password too often.

Beside SSH, there are other protocols supported such as sftp, ftp, dav, etc.

For example, to open a file on a remote FTP server, you could run a command like:

```
vim ftp://hostname/path/to/file
```

Netrw offers lots of options and possibilities for remote editing, so for more information on this, take a look at `:help scp`.

Chapter 7

Personalizing Vim

If you've had any experience with some of the text editors for programmers, it's most likely you'll be disappointed with how Vim looks. But this is actually a good thing. While other editors try to force you use their features, Vim does the opposite.

The "interface" is very minimal. This means that you have to spend some time and effort to make the Vim interface look pretty, as well as to improve your productivity. The benefit is this process of configuration will help you understand better how Vim works.

Vim configuration explained

As a first step, we have to understand how to configure Vim. There are multiple configuration files, which can reside on different locations in your system, depending on which operating system you use or where have you installed Vim.

The main configuration file is `vimrc`. It exists in two versions—global and personal. Your personal `vimrc` file is usually placed in your home directory. In Linux operating systems it's usually a hidden file called `.vimrc`.

Whatever you change in this file will overrule any previous settings in the global `vimrc` file. If you're not sure of your home directory location, run this command in Vim: `:echo $HOME`.

The permanent configuration is set through `.vimrc`. But you can also configure the current Vim session. For example, if you've started Vim, and you don't have line numbers shown, you can run the command:

```
:set number
```

to enable line numbers for the current session. If you'd like to disable this option for the current Vim session, you'd run:

```
:set nonumber
```

Another way to enable/disable boolean options is to use exclamation point `!`. In this case, we could enable line numbers (assuming they're disabled) with a command:

```
:set number!
```

Make Vim look beautiful

Vim allows its users to change the colors it uses. So yes, Vim supports color schemes. To begin, choose some of the installed color schemes. Later you can create your own, or download some you like, and install them in Vim.

In order to choose your color scheme, open a file with some source code. Then type: `:colorscheme` and press *Tab*. Then press *Enter*. You'll see what the scheme looks like. Repeat the same command, just press *Tab* more times, until you find the color scheme you like. Once you find it, add to your `.vimrc` file:

```
colorscheme scheme_name
```

Sometimes, in a big file with lots of code and syntax coloring, it can be difficult to track your cursor. That's why it's a good tip to mark the line the cursor is currently in. You can try this out by typing `:set cursorline` in Vim, or to make this permanent, add to your `.vimrc` file:

```
set cursorline
```

If you don't like the styling of the line, you can change it like this, for example:

```
:highlight CursorLine guibg=lightblue ctermbg=lightgrey
```

If you really have a problem in following your cursor, then you can use a command to mark the current column of the cursor, coloring the entire column: `set cursorcolumn`.

Of course, it's really important to add the line numbers, so also put: `set nu[mber]` to your `.vimrc` file.

If you want to enable spell checking for default, English language, you should add this: `set spell`. If you want spell checking enabled for some other language, you can do it this way (example for German language): `set spelllang=de`. If you want spell checking for more languages at once, no problem: `set spelllang=en,de,it`. Of course, if you change `spelllang` setting to a language that's not installed, Vim will ask you if it should try to download it.

You can always check the configuration of any Vim setting by adding a `?` to the end of its name. For example:

```
set spell?
nospell
```

Usability improvements

Default Vim settings are not really great. If you're going to use Vim seriously, then it's definitely worth it to spend some time on configuration. As we already said, all the configuration we'll manage through the `.vimrc` file.

In this part, I will give you a list of different Vim settings, which you should consider and try out. At the end of this chapter, you'll also find a snippet of basic recommended options, which you can just copy to your `.vimrc` file. Later on, you can continue with configuration on your own.

General configuration options:

- `set nocompatible` - Use Vim settings, rather then Vi settings. It's important to have this on the top of your file, as it influences other options.
- `set backspace=indent,eol,start` - Allow backspacing over indention, line breaks and insertion start.
- `set history=1000` - Set bigger history of executed commands.
- `set showcmd` - Show incomplete commands at the bottom.
- `set showmode` - Show current mode at the bottom.
- `set autoread` - Automatically re-read files if unmodified inside Vim.
- `set hidden` - Manage multiple buffers effectively: the current buffer can be "sent" to the background without writing to disk. When a background buffer becomes current again, marks and undo-history are remembered. See chapter *Buffers* to understand this better.

User Interface Options

- `set laststatus=2` - Always display the status bar.
- `set ruler` - Always show cursor position.
- `set wildmenu` - Display command line's tab complete options as a menu.
- `set tabpagemax=40` - Maximum number of tab pages that can be opened from the command line.
- `colorscheme desert` - Change color scheme.
- `set cursorline` - Highlight the line currently under cursor.
- `set number` - Show line numbers on the sidebar.
- `set relativenumber` - Show line number on the current line and relative numbers on all other lines. Works only if the option above (`number`) is enabled.
- `set noerrorbells` - Disable beep on errors.
- `set visualbell` - Flash the screen instead of beeping on errors.

- `set mouse=a` - Enable mouse for scrolling and resizing.
- `set background=dark` - Use colors that suit a dark background.
- `set title` - Set the window's title, reflecting the file currently being edited.

Swap and backup file options - disable all of them:

- `set noswapfile`
- `set nobackup`
- `set nowb`

Indentation options:

- `set autoindent` - New lines inherit the indentation of previous lines.
- `filetype plugin indent on` - Smart auto indentation (instead of old `smartindent` option).
- `set tabstop=4` - Show existing tab with 4 spaces width.
- `set shiftwidth=2` - When indenting with '>', use 2 spaces width.
- `set expandtab` - On pressing tab, insert 4 spaces.
- `set nowrap` - Don't wrap lines.

Search options:

- `set incsearch` - Find the next match as we type the search.
- `set hlsearch` - Highlight searches by default.
- `set ignorecase` - Ignore case when searching . . .
- `set smartcase` - . . . unless you type a capital.

Text rendering options

- `set encoding=utf-8` - Use an encoding that supports Unicode.
- `set linebreak` - Wrap lines at convenient points, avoid wrapping a line in the middle of a word.
- `set scrolloff=3` - The number of screen lines to keep above and below the cursor.
- `set sidescrolloff=5` - The number of screen columns to keep to the left and right of the cursor.
- `syntax enable` - Enable syntax highlighting.

Miscellaneous Options

- `set confirm` - Display a confirmation dialog when closing an unsaved file.

- `set nomodeline` - Ignore file's mode lines; use vimrc configurations instead.
- `set nrformats-=octal` - Interpret octal as decimal when incrementing numbers.
- `set shell` - The shell used to execute commands.
- `set spell` - Enable spellchecking.

Status line

The statusline in Vim is the bar along the bottom of the Vim window. The purpose of statusline is to give you various information about the status of the current buffer. The default statusline includes info like file path, permissions, line numbers and a percentage number of where you are in the file.

Although the default statusline offers quite a nice set of information, you can always improve it, if desired. There are even a couple of quite popular plugins for this purpose.

We're going to cover just the basics, so if you want to modify your status line, you know how to.

Status line, by default, is shown only if you have more than one buffer open. However, it's better to show it all the time, and you can do this by setting:

```
"show status line
set laststatus=2
```

in your `.vimrc` file. You'll also notice a line `"show status line`, which is a comment describing this option. So whenever you want to add a comment in `.vimrc` file, start the line with `"` character.

If, for some reason, you want to disable it, this is what you need:

```
set laststatus=0
```

Status line can be set like this in your `.vimrc` file:

```
set statusline=%F%m%r%h%w%=(%{&ff}/%Y)\ (line\ %l\/%L,\ col\ %c)
```

This can be a bit hard to read and understand if you're a beginner. A different way of setting it could be something like:

```
set statusline=%t       "tail of the filename
set statusline+=%{&ff}  "file format
set statusline+=%h      "help file flag
set statusline+=%m      "modified flag
set statusline+=%r      "read only flag
set statusline+=%y      "filetype
set statusline+=%c,     "cursor column
set statusline+=%l/%L   "cursor line/total lines
set statusline+=\ %P    "percent through file
```

This format is much more useful, especially if you'd like to experiment with your status line. The easiest way to configure your status line is with the built in flags.

For example, `%m` shows a `[+]` if the current buffer is modified. Using `%L` shows the total number of lines of the current file.

Of course, there are many of them, and that's out of the scope for this book. Take a look at `:help statusline` for more information on them.

> *Related tip:* Using command **g Ctrl-g**, you can show the detailed information about the number of lines, words, characters, etc. in your current buffer.

Swap and backup files dilemma

Swap files

Sooner or later you'll notice that, when you edit files, Vim creates files named like *.filename.swp* in the same location as the file you're editing. These files are called swap files.

Swap files store changes you've made to the buffer. If your Vim crashes, a swap file will allow you to recover those changes. Another important role of swap files is to act as a lock mechanism: if you open a file, which is already opened in another Vim session, you'll be warned. That can be useful, especially on a system with multiple users.

Disabling swap files

You can disable swap files entirely by adding `set noswapfile` to your `.vimrc`. However, I'd recommend you not to disable them, unless you really know what you're doing. Instead, you could organize swap files better.

Swap files organization

Usually the most annoying thing about swap files is that they're created all around your file system, wherever you edit your files. To solve this, you can save all the swap files in one location. Here's how:

1. Create a directory for storing swap files, for example:

```
$ mkdir ~/.vim/swp
```

2. Put this snippet in your `.vimrc`:

```
set directory=$HOME/.vim/swp//
```

The `directory` option contains a list of directories where Vim will try to store swap files. The `//` at the end tells Vim to use the absolute path to the file to create the swap file. This will ensure that swap file name is unique, so there are no collisions between files with the same name from different directories.

Backup files

Vim can make backups of files you edit, so you're safe from losing data. I don't use this Vim feature personally, and I would suggest you set up a better backup solution for your work.

Of course, this feature can be useful. Backups are controlled by the settings of two options: `backup` and `writebackup`. If interested, look these up in `:help`.

Just like for swap files, you can also keep backup files better organized, by creating a directory and adding it to your `.vimrc`:

```
set backupdir=~/.vim/.backup//
```

Project specific .vimrc

If you're working on multiple different projects, with different types of files, you might want to have specific configurations for some types of your projects.

Vim allows you to have a project specific `.vimrc` file. First you need to enable it by adding this to your `.vimrc`:

```
" enable project speficific vimrc
set exrc
```

Then you need to create your specific project `.vimrc` file configuration in the root of your project folder. This way, you can keep your main `.vimrc` file nice and clean, and have a specific configuration for other projects.

Basic recommended configuration

It will take you some time and experience to configure Vim to fit your needs. You can copy this snippet to your `.vimrc` file to get you started. I recommend you start with this minimal (but good!) configuration.

As you progress through the book, your configuration will improve. Most importantly, once you start using Vim regularly, you'll get ideas on what you'd like to improve and change. At that time, come back to this chapter and enable additional options you might need. Also, Google is your friend. Over time, you'll be able to pick up tricks and configuration options from other advanced Vim users.

But take your time, and don't spend a lot of time tweaking your Vim configuration right now. Use this configuration from below, and keep learning.

Here's the basic configuration which you can use right away:

```
set nocompatible        " Use Vim settings, rather than Vi settings
set softtabstop=2       " Indent by 2 spaces when hitting tab
set shiftwidth=4        " Indent by 4 spaces when auto-indenting
set tabstop=4           " Show existing tab with 4 spaces width
syntax on               " Enable syntax highlighting
filetype indent on      " Enable indenting for files
set autoindent          " Enable auto indenting
set number              " Enable line numbers
colorscheme desert      " Set nice looking colorscheme
set nobackup            " Disable backup files
set laststatus=2        "show status line
set statusline=%F%m%r%h%w%=(%{&ff}/%Y)\ (line\ %l\/%L,\ col\ %c)\
set wildmenu            " Display command line's tab complete options as a menu.
```

Chapter 8

Undo and Redo

Vim has a very powerful undo feature. When you press `u` in Normal mode, or run `:u` in Command mode, you'll call the Undo command. To undo all recent changes on the current line, press `U`.

When you press *Ctrl-r* in Normal mode or run `:red[o]`, you'll run the redo command.

If you want to undo multiple times, just press `u` the desired number of times. For example, command `uuu` will undo the last three changes. To undo multiple changes you can also use command `u` with a digit prefix, for example: `5u`—which will undo the last five changes.

That's not all—it gets even better. In Vim, you can travel through time! Using command `ea[rlier]` you can go back in time, while using command `lat[er]` you can travel forward. These two commands work on a state basis. This means that, if you make 4 changes, and run `earlier 2`, last two changes would be reverted. Similarly, if you run command `later 1`, Vim will redo one last change.

The best thing about these commands is that you can actually undo and redo in time frames. For example, if you made a lot of changes in last 10 minutes, and you realized that all of them are wrong, it would take a lot of undo actions. Instead, you can simply ask Vim to undo all the changes you've made in last ten minutes by running a command `earlier 10m`. In similar way you can use the `later` command.

Here are a couple examples which will show you all the possibilities of these commands:

Command	Description
`:earlier 2d`	Undo changes in last two days
`:ea 3h`	Undo changes in last three hours
`:ea 1m`	Undo changes in last one minute
`:later 5m`	Redo all the changes in last 5 minutes
`:lat 15s`	Redo all the changes in last 15 seconds
`:earlier 3f`	Undo last three file states (last three buffer writes)

Undo branches

Vim has one more powerful feature when it comes to undo operation. Let's see an example:

1. Open a new file and write `Hello`. Press *Esc*.

   ```
   Hello
   ```

2. Hit `o` to go to a new line in Insert mode, and write `world`. Press *Esc*.

   ```
   Hello
   world
   ```

3. Now you hit `u`. This undo action will remove word `world`.

   ```
   Hello
   ```

4. Now, as your cursor is on the first line again, on the word `Hello`, press `o`, type `everyone` and press *Esc*.

   ```
   Hello
   everyone
   ```

5. If you hit `u` again, you will undo `everyone`.

   ```
   Hello
   ```

6. If you hit `u` again, you will remove `Hello`. But, you will never get word `world` again. At least in most of the traditional editors.

So, no matter if you made an edit after undoing, you should be able to revert to `world` in the previous example. Vim has a solution for this, and it's called Undo branches. I won't go into details about it, but just tell you how to use this feature.

In the example from above, on step 4, after you complete step 4—and you want to get back `world` again—you need to run command `g-`. Voila :)

Basically, Vim creates an undo branch every time you hit `u`. The branch represents the state of the file before you executed undo. So, you can use `g-` command to move backward or `g+` command to move forward between these branches.

Take a few minutes to experiment with `u`, *Ctrl-r*, `g-` and `g+` and you'll quickly understand how this works. To sum it up: using only `u` and *Ctrl-r* will not get you to all possible text states, while repeating `g-` and `g+` does.

Persistent Undo

All of these feature are great, but there's more! Vim (like every other text editor) can perform undo/redo actions in your current session. Once the session is closed, and you reopen the same file, running undo will do nothing—as you will be already at the oldest change.

Vim supports persistent undo, which means that you can run undo/redo even from your previous sessions. Let's say you edit some file. Then you close it. You open it again. And if you run undo, it will undo the last action from the previous session. This is great feature indeed! This way you can go back historically through changes of any of your files.

How this works? It's simple—Vim creates a new hidden files where it stores the undo history. Now, configuration is very simple. You could add only this line to your `.vimrc`:

```
set undofile   " Maintain undo history between sessions
```

and it would work. However, Vim will write hidden files in the same directory as the file you edit. Over time, this will become messy.

The better way is to create a dedicated directory for these undo history files, running a command like:

```
$ mkdir ~/.vim/undodir
```

My assumption is that `~/.vim` is your main Vim directory. Now, once you have created the directory, you need to add only one more line to your `.vimrc` file:

```
set undodir=~/.vim/undodir
```

That's all. Vim will store all the undo history files in that directory, and you'll have persistent undo working flawlessly. If you want to find out even more about undo feature in Vim, checkout Vim help with `:help undo`.

Chapter 9

Do you speak Vim?

You will love this chapter! You're about to learn a new language—Vim. We can easily adopt Vim elements as language elements. Once you learn basic verbs, nouns and words, you'll be able to do amazing things in Vim, with incredible ease.

Vim Language Elements

We can split Vim elements into three different groups of language elements: verbs, modifiers and nouns. Once you learn these, you'll practically be able to "speak" with your editor.

Verbs

First you need some basic verbs. They can be split in two groups: powerless verbs and powerful verbs.

The powerless verbs is the name I like to use for verbs which can apply only to a single character. That's why they are not really powerful—so, you get where the name comes from.

Here they are:

- `x` - delete character under the cursor to the right
- `X` - delete character under the cursor to the left
- `r` - replace character under the cursor with another character
- `s` - delete character under the cursor and enter the Insert mode

These are useful to know, and you'll probably use them very often.

The powerful verbs, are much more interesting. Here are the most important:

- `y` - **y**ank (copy)
- `c` - **c**hange
- `d` - **d**elete
- `v` - **v**isually select (not really a verb, but used with verbs)

As you can see, it's pretty easy to remember the commands for those words. It's usually the first character of the verb.

Note: These commands are usually known as operator commands or operators.

Modifiers

In Vim, modifiers are used right before nouns, so you can describe how you want to influence the nouns. Here are the most important ones:

- `i` - inner (**i**nside)
- `a` - a (**a**round)
- `NUM` - number (e.g.: 1, 2, 10)
- `t` - searches for something and stops before it (search un**t**il)
- `f` - searches for that thing and lands on it (**f**ind)
- `/` - find a string (literal or regular expression)

Nouns

In English, nouns can be objects you do something to. It's the same in Vim. Here are the most important ones:

- `w`,`W` - start of next **w**ord or **W**ORD
- `b`,`B` - start of previous word or WORD (start of word **b**efore)
- `e`,`E` - **e**nd of word or WORD
- `s` - **s**entence
- `p` - **p**aragraph
- `t` - **t**ag (in context of HTML/XML)
- `b` - **b**lock (in context of programming)
- `h`,`j`,`k`,`l` - left, down, up, right

- `$` - end of line
- `^`, `0` - start of line

These can be expanded and give you even more power:

- `aw` - **a** (complete) **w**ord
- `as` - **a** (complete) **s**entence
- `ap` - **a** (complete) **p**aragraph
- `iw` - **i**nner **w**ord
- `is` - **i**nner **s**entence
- `ip` - **i**nner **p**aragraph

Learn to talk to Vim

Now that we've covered basic language elements, we can start learning how to talk to Vim. You will be able to "talk to Vim" in sentences. Let's see some examples:

- Delete the current word: `dw` (**d**elete **w**ord from cursor position to the end of the word)
- Change current sentence: `cis` (**c**hange **i**nside **s**entence)
- Change a string inside quotes: `ci"` (**c**hange **i**nside quote)
- Change until next occurrence of 'hello': `c/hello` (**c**hange search hello)
- Change everything from here to the letter Y: `ctY` (**c**hange un**t**il Y)
- Visually select this paragraph: `vap` (**v**isual **a**round **p**aragraph)

Change inside quotes

For example, if you have a line:

```
print "Hello world!"
```

Placing your cursor on the line, and running `ci"` says to Vim: change inside quotes. You'll get:

```
print ""
```

with cursor placed between `""`, in Insert mode. This way, you can easily change content between the quotes. You could also run delete instead of change, by running `di"`.

Change inside tags

Let's say you have an HTML file open, with a line:

```
<h1>Welcome to my site</h1>
```

You want to change the content of this tag. You could say to Vim: **d**elete **i**nside **t**ag, or **c**hange **i**nside **t**ag. In this case, you want to tell Vim: change inside tag.

You’ll do this by positioning your cursor on the specific line mentioned above (at any character in the line). Then, in Normal mode run `cit`.

Result will look like this:

```
<h1></h1>
```

with cursor positioned in a between tags, in Insert mode.

If you take a look at the last two examples, you’ll see the pattern—both sentences have the structure like: `<verb><modifier><noun>`.

More examples

Note: While you’re reading description, try to guess what the command would be, before actually seeing it.

- **D**elete a line: `dd`
- **D**elete to the next **w**ord: `dw`
- **D**elete the whole current **w**ord: `daw`
- **D**elete up un**t**il the next comma (**,**) on the current line: `dt,`
- **D**elete to the **e**nd of the current word: `de`
- **D**elete to the **e**nd of next **w**ord: `d2e`
- **D**elete down a line (current and one below): `dj`
- **D**elete up un**t**il next closing parenthesis: `dt)`
- **D**elete up until the first search match for “rails”: `d/rails`
- Jump **3 w**ords from cursor forward and **d**elete next **2 w**ords: `3wd2w`
- **D**elete a word from cursor to beginning of a word, including the character under cursor: `dvb`

For more commands take a look at `:help motion.txt`.

The “dot” command

I believe you have already heard of the principle *Don’t Repeat Yourself*. In software engineering, this is a principle of software development where your focus is on reducing repetition of all kinds. As you’ll see throughout the book, Vim has many ways and commands to automate different kinds of tasks, so you don’t have to repeat your actions.

One of the most powerful Vim command when it comes to avoiding repetition is the `.` (“the dot”) command.

Hitting `.` in Normal mode will repeat the last native Vim command you’ve executed.

Let’s say you want to delete 5 words from the cursor forward. As you already know, you could press `5dw` and it’s done. However, sometimes it’s not convenient to mentally count the number of words. An alternative would be to use `dw` to delete one word. And then press `....` to call the dot command four times. In this case, you would repeat the latest, `dw` command, four more times, and in this way achieve the same effect without counting the words.

If you used `dd` to delete a line, and you want to delete 4 more lines, you could also execute `4.`, instead of pressing `.....`. That also works.

It’s very important to understand what is actually repeatable by the dot command.

For example, if you have a sample code like this:

```
my $i
my $learn
my $quickly
```

and your cursor is positioned on the first line. You want to append `;` to all three lines.

You could run a command like:

```
A;<Esc>j
```

- `A` - would place your cursor at the end of the first line in Insert mode.
- `;` - you press to actually insert it, and then you press *Esc* to get back to Normal mode.
- `j` - to move one line down

Now, your cursor is at the second line. If you then press `.` to repeat the change in next (second) line, this won’t work. Here’s what you’d get:

```
my $i;
my $learn;
my $quickly
```

Your cursor will still be at the second line rather than on the third line, but `;` will be appended. This brings us to conclusion that only this part of our original command was repeated: `A;`*Esc*.

Now, why is this the case? It’s important to remember that with the dot command, you can repeat the commands which change the contents of the buffer. A **change** is any command which you can use to modify your text. In our example, we had the case that command `j` wasn’t repeated, and our cursor wasn’t moved to the third line. Commands like `j` are called motions (or nouns, as we called them previously in this chapter)—and they don’t affect the text itself. Command `j` just moves the cursor, but doesn’t change text anyhow, so it can’t be repeated.

Think in terms of the grammar of your native language: Not nouns, but verbs are used to express some sort of action. The same is true in Vim: nouns (or motions) can't affect the text, so they can't be repeated with the dot command.

Of course, if you'd like to repeat multiple changes, or a combination of movements and changes, you can easily record those into a macro. You'll learn more on macros later. To see all the commands which can affect the text in a buffer, take a look at `:help change.txt`

Chapter 10

Substitution

Ranges

For most Vim commands, the default range is the current line. This means that action performed by a command will affect only the current line. However, you can control ranges, and in that way execute commands over the custom ranges of lines or characters in the current buffer.

For example:

- `:s/bad/good/g` - changes all words `bad` to `good` in the current line.
- `:6,11s/bad/good/g` - makes the same change, but in lines 6 to 11, including 6 and 11.
- `:%s/bad/good/g` - makes the same change in entire file.

Let's analyze these examples.

As you remember, `s` command stands for `substitute`. Since we're doing search and replace (substitution), that explains the `s`. If we remove the part of the string which is the same in every example `/bad/good/g`, then remove `s` and we remove `:` (which we always use to run a command), here's what's left:

- ' ' - First example: nothing is left, so range is not defined. This means that default range is applied, which is usually the current line.
- `6,11` - Second example: It's pretty obvious that this is how we defined a range of lines from 6 to 11.
- `%` - Third example: This is a special character to define the whole file.

How to define ranges

If line range is not specified, the `substitute` command will operate on the current line only, by default. As already mentioned, for most commands, the default range is `.` (the current line). However, for `:g` (**g**lobal) and `:w` (**w**rite) commands the default is `%` (all lines).

Range	Description	Example
`28`	line 28	`:28s/bad/good/g`
`1`	first line	`:1s/bad/good/g`
`$`	last line	`:$s/bad/good/g`
`%`	all lines in a file (same as `1,$`)	`:%s/bad/good/g`
`6,28`	lines 6 to 28 inclusive	`:6,28s/bad/good/g`
`11,$`	lines 11 to end of the file	`:11,$s/bad/good/g`
`.,$`	current line to end of the file	`:.,$s/bad/good/g`
`.+1,$`	line after current line to end	`:.+1,$s/bad/good/g`
`.,+4`	current to current+5 line, inclusive)	`:.,.+4s/bad/good/g`
`?a?,/b`	between patterns a and b, inclusive)	`:?a?,/b/s/bad/good`

Search and replace

The best way to understand powerful Vim features when it comes to search and replace is to learn through examples.

As we already saw from the previous example, the command for text substitution in Vim can be defined like:

```
:[range] s[ubstitute]/pattern/string/[flags] [count]
```

Everything enclosed between `[ ]` in this command is optional.

Substitution flags can be:

- `c` - to **c**onfirm each substitution
- `g` - to replace all occurrences in the line
- `i` - **i**gnore case for the pattern
- `I` - don't ignore case for the pattern

Before we go into the details, let's cover some basics. Once you start the search and get the first matched string, in order to jump to the next match, press `n`. To get back to previous match, press `N`.

Replacement in the whole file

Let's say your task is to replace all occurrences of the string with new string in the whole file. This is a very common use case. Here's how you do it:

```
%s/old_string/new_string/g
```

- `%s` means that substitution will be performed for all the lines.
- `g` specifies that the action will be performed over all occurrences in the line.

So with this command we basically told Vim to replace `old_string` with `new_string` for every occurrence in a line, for all lines in a file.

If the `g` flag were not used in the command above, only the first occurrence in the line would be replaced.

Replacement within a single line

To achieve this, you need to run:

```
:s/old/new/gi
```

You could leave out `i` at the end, but in this case we get case insensitive search. If you'd like to perform a case sensitive search, you should use the `I` flag instead of `i`.

Replacement within a range of lines

You can perform substitution over a range of lines you define. In this example, we will replace `old` with `new` on lines 15 to 30:

```
:15,30s/old/new/g
```

Replacement inside visual selection

You can perform search and replace on your visual selection. While in Normal mode, use `v` to visually select lines where you want to search and replace. Then, type `:`, which will automatically change to `:'<,'>`. Now you can start typing the rest of substitution command, like:

```
'<,'>s/helo/hello/g
```

Replace only the whole words

Sometimes you'll want to replace the string which is only a whole word, and not when it's a part of some longer word.

For example, you want to substitute word `is` with `was` from the sentence below:

```
This sentence is short.
```

The usual substitution command would be:

```
:s/is/was/g
```

Simple, right? But look at what we would get:

```
Thwas sentence was short.
```

Something's wrong, isn't there? The string `is` in word `This` was also replaced by `was`.

In order to search and replace only the whole words, you need to use `\<` and `\>` to specify that only whole word should be matched. Here's how the correct command would look:

```
:s/\<is\>/was/g
```

Now the result will be correct:

```
This sentence was short.
```

Replace either string1 or string2 with a new word

Using regular expression, we will replace words `pretty` and `good` with `awesome` in this sentence:

```
That pretty girl did good on test.
```

Running the following command:

```
:s/\(pretty\|good\)/awesome/g
```

and you should get a result like:

```
That awesome girl did awesome on test.
```

One new character you will notice in this command is `|` which is "logical OR". This way you tell Vim to find the first word OR the second word and replace it. But in order to use it, you have to add `\` in front of it. So when you want to use logical OR, you should always use `\|` in the substitute command. If you find this interesting, take a look at `:help magic`.

Interactive search and replace

Using `c` flag, you can run interactive search and replace in Vim.

Let's see an example:

```
:%s/bad/good/gc

replace with good (y/n/a/q/l/^E/^Y)?
```

This command is already familiar to you. It searches for string `bad` and replaces it with `good` in entire file.

But, since we have added the `c` flag for confirmation, Vim will ask us what we want to do, for each match it finds.

This means that for every match you have to type a letter and tell Vim what you want to do. Here are the options:

- `y` - **y**es - will replace the current highlighted word. After replacement of this word, Vim will highlight the next word that matched the search pattern.
- `n` - **n**o - will not replace the current highlighted word. However, it will highlight the next word that matched the search pattern.
- `a` - **a**ll - will replace all of the remaining matches without asking you what to do.
- `q` - **q**uit - will simply quit the replacement process.
- `l` - **l**ast - will replace only the current highlighted word and terminate the replacement process.

Here `^E` is actually *Ctrl-e*, and enables you to scroll the screen up. Similarly, `^Y` or *Ctrl-y* enables you to scroll the screen down. However, these commands might not be available if your Vim wasn't compiled with `insert_expand` support.

> *Related tip:* In order to check the details about your Vim version, you can run a command **:version**. You'll be able to see different details, including the information if **insert_expand** option in enabled. If yes, there will be a + sign in front of it.

Search through multiple files

You can easily search through multiple files using the `vimgrep` command. This command will search for the string `warning` in all markdown files (ending with `.md`) in the current directory:

```
:vimgrep warning *.md
```

This command will jump to the first file that contains a match (by default).

Here are the commands you can use to navigate through search matches:

- `:cn` - jump to the next match.
- `:cN` - jump to the previous match.
- `:clist` - view all the files that contain the matched string
- `:cc number` - jump to specific match number, which you get from `:clist` output.

It's also possible to search recursively. Let's say you want to search for the string `error` in all `*.log` files under the current directory and all subdirectories. Here's what you need to do:

```
:vimgrep error **/*.log
```

Match that string

If you'd want to highlight all matches of a string in your current buffer, you can use the `:match` command. Let's assume you're looking at some log file in Vim, and you want to highlight all the occurrences of string `Error` in a red color (the color depends on your color scheme, and can be different than red):

```
:match ErrorMsg /Error/
```

Let's see what we have here: - `:match` - the command itself - `ErrorMsg` - predefined color in Vim for error messages - `/Error/` - search pattern we defined

Here's the list of predefined colors you could use:

- `ErrorMsg`
- `WarningMsg`
- `ModeMsg`
- `MoreMsg`

You can even predefine your own color.

Related tip: You can always show your next matched string at the center of the screen when you press **n** or **N**, so it is easier to identify your location in the file. To enable this, put the following lines in your .vimrc:

nnoremap n nzz

nnoremap N Nzz

The power of the global command

The global command `:g[lobal]` is very useful. The way to use it is presented below:

```
:[range]g/pattern/cmd
```

Let's break it down:

- `[range]` - We have already covered it with the **s**ubstitute command. It's optional to provide a range, but it's important to notice that the default range is the whole file.
- `pattern` - is the pattern we're looking to match in the file
- `cmd` - is the Ex command to be executed for each line matching the `pattern` (reminder: an Ex command is the one starting with a colon such as `:d` for delete).

Let's see some examples.

Delete all lines containing a pattern

We want to delete all lines in a current file which contain the string `error`:

```
:g/error/d
```

- `g` - the **g**lobal command itself
- `error` - the pattern we want to match
- `d` - the Ex command **d**elete

Delete all lines not containing a pattern

We want to delete all lines not containing the string `important`:

```
:g!/important/d
```

The only difference from the previous example is that we added `!` after `g`. The `!` tells Vim to invert the matching.

Delete all blank lines

We want to delete all blank lines in the current file:

```
:g/^\s*$/d
```

Let's break down this command:

- `g` - the global command
- `^` - the start of line
- `\s*` - zero or more **s**pace characters
- `$` - the end of line
- `d` - the Ex command **d**elete

So we basically tell Vim something like: In the entire file, delete all lines which contain zero or more space characters.

Delete huge number of lines

As we already saw, the command `:g/pattern/d` performs delete on all lines matching the pattern. When a line is deleted, it is first copied to a register. Since no register is specified, the default (unnamed) register is used.

Tip: You might want to check out the Registers chapter.

If you try to delete many thousands of lines, the copy process can take some time. So in order to avoid that waste of time, you can use the blackhole register. The blackhole register `_` can be specified because any copy or cut into the blackhole register performs no operation.

Finally, to delete all of lines matching a pattern, very fast, this is the command:

```
:g/pattern/d_
```

Execute macros with the global command

Let's say you have recorded a macro in register `a`, and you want to execute the macro on every line that contains string `vim`. You can do this using `:g`, together with `:normal` command:

```
:g/vim/normal @a
```

- `:g/vim/` - matches all lines with string `vim`.
- `normal @a` - tells Vim, execute macro stored in register `a`, in Normal mode.

Don't worry if you don't understand this now. You're going to learn more about macros soon.

Copy and move lines using :g

You can even copy or move lines within your file using the `:g` command. Here's how to copy lines 3 through 9 after line 20. It doesn't matter where your cursor is:

```
:3,9t20
```

You can remember it as: lines 3-9 **t**o line 20. If in the command above you used `$` instead of `20`, lines 3 to 9 would be copied to the end of the file.

So you'll use `t` for copying the lines with `g` command. But if you want to move the lines, then you need to use `m`. This means that, if you'd like to move lines 3 to 9 to line 20, you'd need to run:

```
:3,9m20
```

You can also work with the current line instead of ranges. Here's how to move the current line to the top of your file (to line 0):

1. Place the cursor on the line you want to move to the top of the file.

2. Run this command: `:m0`

For every line containing “good” substitute all “bad” with “ugly”

The title says it all. Here’s how to do it:

```
:g/good/s/bad/ugly/g
```

Let’s break it down:

`:g/good/` - this part is matching lines containing word `good` in the entire buffer.

`s/bad/ugly/g` - is a substitute command which replaces `bad` with `ugly` in all occurrences, on previously matched lines (with the first part of the command).

Reverse all the lines

Let’s say you have a file like:

```
first line
second line
third line
```

and you would like to reverse the order of lines, so you get:

```
third line
second line
first line
```

You could achieve this by executing the following command:

```
:g/^/m0
```

For more info on this one see `:help 12.4`.

For more information on the `global` command in general, check `:help 10.4`.

Chapter 11

Registers

This entire chapter is like one big example, consisting of many smaller examples. Read this chapter at least twice, and make sure that you go through examples in Vim.

Warning: *This is a chapter where you might get stuck. It requires your full focus. So don't rush. Don't go over it in a hurry. Give yourself time, and if possible, read it at the period of the day when your concentration is good.*

As a software developer you probably have to perform actions like copying different pieces of your code from multiple files into different locations of your current Vim session. Using only the system clipboard, this can be a cumbersome and time consuming task. Once you master Vim registers, your text editing efficiency will greatly improve.

A register is a kind of clipboard. It is a place in memory that you can use for storing text. Vim supports several kinds of registers and it fills some of them automatically when you perform actions like yanking or deleting text. Others can be filled explicitly by the user.

It's possible to be productive in Vim without knowing much about registers, but if you really want to edit text efficiently, you have to understand how they work.

Think of registers as different buffers for text. Most operating systems and applications have only a single clipboard that you can use for copy, cut and paste operations. Vim is different. In Vim you have not one or two but nine different types of clipboards!

Yes, there are nine types of registers. Their names start with " like register `"a`. Here's a list of all register types:

- The unnamed register `""`
- Ten numbered registers `"0` to `"9`
- The small delete register `"-`
- Twenty-six named registers `"a` to `"z` or `"A` to `"Z`
- Four read-only registers `":`, `".`, `"%` and `"#`
- The expression register `"=`
- The selection and drop registers `"*`, `"+` and `"~`
- The black hole register `"_`
- Last search pattern register `"/`

Don't try to remember all of them at the moment.

Using Vim Registers Internally

In order to understand registers better, we'll go through a simple example.

Let's imagine this scenario: you have installed the newest version of PostgreSQL server on your machine, which comes with a default configuration file. You need to copy a few configuration parameters from your old PostgreSQL configuration file to the new one.

The problem is that both configuration files are big, and we'll have to go back and forth between old and new files multiple times in order to copy everything we need. Fortunately, we can use the power of registers to make this operation more efficient.

For simplicity, let's say there are only three lines of configuration parameters we want to copy from the old to a new configuration file, and they are separated only by one empty line. Here's the sample content of old configuration file:

```
1 port = 5432
2
3 work_mem = 512MB
4
5 wal_level = hot_standby
```

Our main point is to go through the old configuration file only once, and copy these three lines of text into three different registers.

In Normal mode, you need to yank the first line to register `"a`. You can do this as usual, by yanking an entire line with `yy` command, but with a small addition—you need to tell the `yy` command where to yank it to. That's why you need to run `"ayy` command. Similarly, yank the third line to register `"b` using `"byy` command. Finally, yank the fifth line to register `"c` using `"cyy` command.

At this stage, we have copies of three lines from the snippet above, stored into three different named registers: `"a`, `"b`, and `"c`.

Now, in the new PostgreSQL configuration file, we want to paste these three lines to appropriate locations:

In Normal mode, you'd find the appropriate location for line `port = 5432` which is now stored in register `"a`, and paste it using `"ap` (paste after cursor) or `"aP` (paste before cursor) command. Then, to paste line `work_mem = 512MB` stored in register `"b`, you can similarly run `"bp` or `"bP` commands. Finally, to paste the last yanked line `wal_level = hot_standby` which is stored in register `"c`, run `"cp` or `"cP` command.

You have just used three different clipboards in Vim. It is as simple as that!

To be completely correct, you have just used `"a`, `"b` and `"c` named registers. They are just three of 26 named registers: `"a` to `"z` or `"A` to `"Z`. Vim stores text in those registers only when you explicitly tell it to. You might think that there must be actually 52 registers, but there's an important difference between registers starting with a lower-case letter and those starting with an upper-case letter.

For example, `"a` and `"A` are pointing to the same register. If you yank some text to register `"a` using `"ay`, the previous content of register `"a` gets replaced by the text that has been yanked. If you yank text using `"Ay`, the previous content won't be replaced. Instead the newly yanked text will be appended to the register's content.

That's all you need to know about named registers. For your information: I have never used more than four named registers at the same time. However, if you forgot which register you used, simply run the `:registers` command and you'll get the preview list of your registers.

Of course, instead of yanking text you can also delete text and put it into a certain register using `"ad`. To cut the text and store its content in register a use `"ax`.

Whenever you yank (`y`), change (`c`), delete (`d`), substitute (`s`) or cut (`x`) some text in Vim, it's copied to the unnamed register. Vim uses the content of the unnamed register for any put (paste) command (`p` or `P`) that doesn't specify a register explicitly.

So if you, for example, use command `"ad` and then paste using the `p` command (without selecting any register), you'll paste the text stored in the unnamed register `""`.

The famous annoying problem finally solved

There's a problem that annoys many Vim users, which can be solved easily using registers. Here's an example of it:

You delete some text with `d` (and not specifying a named register) which you want to paste to other location. Before pasting your text to the new location, you have to delete some more text in between.

As we already stated, modifications using the `d` command will go into default `"` register, so you'll lose your original text from step 1.

To solve this problem you need to know that Vim saves your last yank in the 0 register automatically. This means that you can yank text (without using a named register), then delete some text in between, and eventually paste the previously yanked text with `"0p`.

Vim also gives you the possibility to easily enter the same piece of text multiple times, without copy and paste operations. You can achieve this using “the dot” command. As mentioned before, the dot command `.` is very useful. What we didn’t mention is that when using the dot command you can access the `".` register, which contains the last edit made in Insert mode. Here’s a simple example:

You’re in Insert mode. You need to create a list of 10 local IP addresses. You type text: `192.168.1.10` on the first line, press `Enter` to move to next line and then press `Esc`. Now in Normal mode, press `.`. This command will insert the same text `192.168.1.10`. You can repeat this command, and invoke `.` eight more times, and you’ll get a list of ten IP addresses (all the same, but easy to modify). Alternatively, after the first step, you can press `9.` and insert the text 9 more times.

So just by invoking `.` command, you are able to paste the content of `".` register. This Vim register is one of four read-only registers.

The System Clipboard Registers "+ and "*

All the registers mentioned so far are internal Vim registers. That is you can use them only within Vim. However, when you want to copy some text from Vim to another application on your system (or vice versa), you need to use the `"+` or `"*` registers.

To copy some text from Vim to the system clipboard use the `"+` register. Let’s say you want to copy text from lines 1 to 3 of the following code to some other application such as Libre Office.

```
1 my $ceph     = get cmdb 'ceph';
2 my $hostname = run 'hostname';
3 my $config   = $ceph->{$hostname};
4
5 file '/etc/ceph/ceph.client.admin.keyring',
6  source => 'files/etc/ceph/ceph.client.admin.keyring',
7    owner  => 'ceph';
```

As you’ve already learned, you can copy lines 1 to 3 by placing your cursor on line 1 and invoke the `3yy` command. In this case, we need to yank these three lines to the system register `"+`.

Here’s what you have to do:

1. Position your cursor on line 1, then use `"+3yy` to yank lines 1 to 3 to the system register `"+`.
2. Go to the application you’d like to copy the text to (for example Libre Office) and press *Ctrl-v* to paste it. You do not have to necessarily use *Ctrl-v*. You can use any other way of invoking a paste command.

This works on Linux, Windows, and macOS. However, when you need to copy text from another application to Vim things get a little more complicated. If you're using Vim on Windows or macOS both the `"+` and the `"∗` register refer to the same system clipboard, so it doesn't matter which one you'll use.

In the example below, we'll copy a URL from a web browser into our Vim session:

1. Select the URL in your browser and press *Ctrl-c* to copy it to the system clipboard. Also, you could use any other way your browser supports to copy the URL.
2. In your Vim session enter Normal mode and press: `"∗p` or `"+p` to paste the URL.

Note that it doesn't matter if you use `"∗` or `"+` in step two.

When you're running Vim on Linux, there's something more you need to know in order to work effectively with the system clipboard: X11 window system (present on your system) has three different system clipboards. We care only about two of them here.

If you're using any application other than Vim, and you copy text using *Ctrl-c*, *Ctrl-x* or any similar action, this content is stored in a system clipboard called CLIPBOARD. This is generally the main system clipboard. In this case, when you want to paste that content to Vim, you need to use the `"+` register.

For example:

1. Open a random web page in your browser and select a paragraph of text. Then press *Ctrl-c* (or copy the text using a command your browser supports).
2. In Vim, while in Normal mode, use `"+p` command to paste that text.

However, when you only select a text in your terminal, web browser or any other application on your system, this text is copied to the second X11 system clipboard called PRIMARY. So when you want to paste this text in Vim, you need to use register `"∗`.

For example:

1. Use your mouse, touchpad or keyboard to select only one sentence on a web page in your web browser. Don't do anything else.
2. In Vim, while in Normal mode, use `"∗p` command to paste that sentence.

In all of the examples above the paste operations happen in Normal mode. Don't forget that you can always paste in Insert mode as well, using *Ctrl-r* `+`.

Note: If the two examples above don't work in your Vim, it doesn't support the X11 features mentioned. You can check this by running the `:version` command. If there's a `+` sign in front of `xterm_clipboard`, Vim supports this feature. If your Vim doesn't support it, you can recompile Vim and enable it.

Chapter 12

Buffers

One of the most confusing features of Vim for beginners are buffers. A buffer is a piece of memory that's been loaded with contents of a file. So when you open an existing file called `mastering_vim.txt`, the contents of this file is now loaded in one Vim buffer.

Very often, beginners don't really understand the difference between buffers, windows and tabs. That's why it's important to define it right away:

- A buffer is the in-memory text of a file.
- A window is a viewport on a buffer.
- A tab page is a collection of windows.

No matter how you edit this file in Vim (you're actually editing the contents of a buffer), nothing will be written to the original file on disk until you actually write to the buffer.

What's confusing for users of other text editors is that buffers can be hidden. Those users got used to using tabs. You close a tab, and the file is closed. Not in Vim. Vim gives you power to have only one tab, but many buffers open.

Now, how do you use buffers? It's very simple actually.

For example, run `:ls` command to list the buffers. Or you can use `:buffers` command for the same result. In order to select the buffer you want to go to, you need to press the number under the desired buffer shown in the list, followed by `b`. For example, to switch to buffer 5, you'd run a command `:5b`.

If you already know the buffer number in the list, you can go directly to that buffer by running `:buffer N` where N is the number of the buffer.

To open all of the buffers in windows, type `:ball`. Remember it as "buffers all". There's also `:bnext` and `:bprevious`, or short form `:bn` and `:bp` so you can cycle through them. If you want to close the current buffer, just run `:bd`. Remember it as "buffer delete".

You get the idea that this is not so handy. In order to make buffer cycling much more effective, you can add the following mapping to your configuration file:

```
map <C-K> :bprev<CR>
map <C-J> :bnext<CR>
```

With this configuration, you'll be able to go to the previous buffer by using *Ctrl-k* and go to next buffer with *Ctrl-j*. It feels much better!

If you want to switch back and forth between your current and previous buffer, you can press `Ctrl-6`.

You'll understand mapping better once you go through the *Mapping* chapter.

Chapter 13

Windows, Tabs and Sessions

Windows and Tabs

As we already said, a buffer is the in-memory text of a file. Official documentation says that a window is a viewport on a buffer, while a tab page is a collection of windows.

You already know that a buffer is a file loaded into memory for editing. The original file remains unchanged until you write the buffer to the file.

A buffer can be in one of three states:

- active - when the buffer is shown in a window.
- hidden - when the buffer is not shown in a visible window.
- inactive - when the buffer is not shown, and it doesn't contain anything.

You can use multiple windows to display one buffer (one file content). Also, you can use several windows to display a few different buffers.

By default, Vim starts with one window.

You can use `-o` and `-O` arguments to open a window for each file in the argument list. The `-o` argument will split the windows horizontally, while the `-O` argument will split the windows vertically.

For example, running this will open three windows, split vertically: `vim -O file1 file2 file3`. You could also use `-oN`, where N is a decimal number, to open N windows split horizontally.

If there are more file names than windows, only N windows are opened and some files do not get a window. Similarly, `-ON` opens N windows split vertically, with the same restrictions.

Split Windows

A window can be split, vertically or horizontally. Let's say you've opened one file in Vim.

Now you want to open the `second_file.txt`. Here's what you can do:

- `:sp[lit] second_file.txt` - opens file as a horizontal split
- `:vs[plit] second_file.txt` - opens file as a vertical split

Both `:sp` and `:vsp` can take a numerical value, to define the split size.

For example, this will create a horizontal split to show 20 lines of `third_file.txt`:

```
:20sp third_file.txt
```

The numerical value for vertical splits represents the character width of the column, for example:

```
:25vsp third_file.txt
```

Switching windows

Here are the default shortcuts for switching windows:

Shortcut	Description
Ctrl-w h	Switch to the window to the left
Ctrl-w j	Switch to the window below
Ctrl-w k	Switch to the window above
Ctrl-w l	Switch to the window to the right

However, if you'll use them often, you'll notice they're not very handy. Here's a suggestion on how to make these operations more smooth:

```
nnoremap <C-H> <C-W><C-H>
nnoremap <C-J> <C-W><C-J>
nnoremap <C-K> <C-W><C-K>
nnoremap <C-L> <C-W><C-L>
```

We can use different key mappings for easier navigation between splits. If you put this snippet in your `.vimrc`, you could, instead of *Ctrl-w* then `j`, switch to window below just by *Ctrl-j*.

Moving windows

Very often you'd like to move your windows around. Here are some of the default shortcuts to move your windows around:

Shortcut	Description
Ctrl-w r	Rotate the windows from left to right (only if they are split vertically)
Ctrl-w R	Rotate the windows from right to left (only if they are split vertically)
Ctrl-w H	Move current window to the far left and use the full height of the screen
Ctrl-w J	Move current window to the far bottom and use the full width of the screen

Resizing windows

Very often you'd like to resize your windows. Here are some ways to do it:

Shortcut	Description
Ctrl-w =	Resize the windows equally
Ctrl-w >	Incrementally increase the window to the right (takes a parameter: `Ctrl-w 10 >`)
Ctrl-w <	Incrementally increase the window to the left (or `Ctrl-w 10 <`)
Ctrl-w -	Incrementally decrease the window's height (or `Ctrl-w 5 -`)
Ctrl-w +	Incrementally increase the window's height (or `Ctrl-w 5 +`)

One more great tip you should know is about the `:on[ly]` command. If you have a few splits open, but you only want one window in focus, you may find it tedious to navigate to those other split windows to close them. Of course, there is an easier way. The Ex command `:on[ly]` will close all other splits, excluding the window that is currently in focus. See `:h :only` for more details.

For more information on splits, check `:help splits`.

Sessions

If you use split windows and tabs a lot in your workflow, you know that it can be painful (or at least boring) to every single time create and position your windows and tabs, the way you want.

Vim offers a great solution—sessions! With sessions, you can save the position of your windows and tabs once you're done with work. When you come back to work and start up Vim, it's very

fast and easy to restore the previous session.

So once you have the session you'd like to save, it's enough to run a command like this: `:mksession ~/mysession.vim`.

Of course, you can select a different path and filename for a session. Then, after you exit Vim, and you want to restore your saved session, you can do either of the following:

- In terminal, you can run `vim -S ~/mysession.vim` and start Vim with your session already loaded.
- Start Vim, and run command: `:source ~/mysession.vim`

Chapter 14

Macros

Macros represent a simple concept which can be described as “record the sequence of my actions, save them, and anytime I need them again, execute them.” It can be a very powerful automation tool.

This is probably the most underused feature which can improve your productivity dramatically. I find macros especially useful when I need to make some formatting changes to multiple lines.

Basically, almost everything you’re doing in Vim in some text can be recorded, and then just executed on the other text, without repeating all the actions.

Let’s see an example workflow:

1. From this moment, start recording my actions into register `a`: `qa`. Any register can be used.
2. Press `q` during recording to stop it.
3. Execute the recording in register `a` you’ve just made by pressing `@a`.
4. To repeat the last executed macro, you can simply run `@@`.

Of course, you can insert any number before the `@` to repeat the execution of macro that number of times. For example, `10@a` will execute macro recorded in register `a` 10 times.

> *Related tip:* If you run a macro, even the simple one, over a few thousands of lines or more, its execution can get pretty slow. This is the case because with each macro execution Vim tries to redraw the screen, which is an expensive operation. That’s why, if you get into a similar situation, you should enable the option **lazyredraw**.

Execute macro in multiple files

Sometimes you'll need to run a macro in more than one file. Once you have recorded a macro, you need to build an arguments list with the files.

For this example we will load all models from a Rails application:

```
:args app/models/*.rb
```

We assume that a previously recorded macro is in register `a`. We run the macro with:

```
:argdo normal @a
```

All you have to do now is to save all the buffers with the command:

```
:argdo update
```

Note: The command `normal` is used to emulate executing commands in Normal mode. More on this command in chapter *Effective multiple file editing*.

Editing a macro

Sometimes you'll make a mistake with a longer macro, and you would rather edit it than record it again from scratch. Here's how you can do this:

1. Insert the macro on an empty line by running a command: `"<register>p`. If your macro is saved in register `m`, the command would be `"mp`.
2. Edit the macro the way you want.
3. Copy the macro into the correct register: move the cursor to the beginning of the line and run `"my$` (where `m` is the correct register). You can again use the original register, or use another one. If you want to use register `n`, the command would be `"ny$`.

Recursive macros

Using macros can be even more effective with recursion. Recursive macros are especially useful when you need to act on every line in the file. In order to record a recursive macro, you need to start with an empty register.

> *Related tip:* To empty a register, press **q<register>q**.
>
> For example, to make empty register **a** press **qaq**.

Start recording your macro, as shown before. Just after you recorded the last action of your macro, invoke the macro itself as the last command. As we made sure that the register is empty, calling a macro won't do anything.

Let's take a look at the sample text:

```
mastering
vim
quickly
wtf
omg
.
in
.
no time
```

1. Place the cursor on the first line. We assume that register `a` is empty, so we could record a macro like this: `qaI"`*Esc*`A"`*Esc*`j@aq`
2. Step by step: you'd press `q` followed by `a` to start recording a macro in register `a`.
3. Then you'd press `I` which would put you in Insert mode (at the beginning of the line), and then you'd type `"`.
4. Hitting *Esc* will get you back to Normal mode. Then you press `A`, which is the command for appending the text. You end up in Insert mode again, and you append `"` to the end of the word in the current line.
5. You hit *Esc* one more time to get back to Normal mode, then press `j` to move to the next line.
6. And here we come to the key part: you finally execute the current macro itself—everything you recorded so far—to be applied on the line you just moved to. So you press `@a` to call the macro, and then `q` to stop recording it.

Then with a single invocation of `@a`, all the lines of the file would be now inside the double quotes.

More macro examples

You can use `norm[al]` command in Ex mode to help you increase your efficiency with macros.

Here are a couple of examples of how to use it:

- Execute the macro stored in register `v` on lines 6 through 16: `:6,16norm @v`
- Execute the macro stored in register `i` on lines 10 through the end of the file: `:10,$norm @i`
- Execute the macro stored in register `m` on all lines: `:%norm @m`
- Execute the macro stored in register `o` on all lines matching pattern: `:g/pattern/norm @o`
- Execute the macro on visually selected lines: select lines in Visual mode. Then type `:norm @a` and observe that the following input line is shown: `:'<,'>norm @a`. Hit *Enter* and macro recorded in register `a` will be executed.
- Execute the macro in register `a` to the next 100 lines, if macro itself moves to the next line: press `100@a`.
- Execute the macro in register `a` to the next 100 lines, if macro itself doesn't move to the next line: press `.,+99norm @a`.

Note: For more details on `:normal` command, see the chapter *Effective multiple file editing*.

Chapter 15

The power of Visual modes

Vim allows you to visually select the content and manipulate the selection. The usage of Visual mode can be described in three steps:

1. Start selecting your text with `v`, `V` or *Ctrl*-*v*. The character under the cursor is used as the beginning of selection.
2. Move to the end of the text you want to select. The text you've selected is highlighted.
3. Type a command to execute over the highlighted text.

Running command `gv` in Normal mode will re-select the previous visual selection. This is handy when you're performing a search and replace on a visual selection and don't match the substitution quite right on the first try.

Selecting characters

You can select characters in a line using command:

```
[count]v
```

Pay attention here, this is tricky.

For example, if you'd like to select 3 characters, the one under your cursor, and two after it, you could press: `3v`. As you already know, `[count]` is optional, so you can simply use `v` to start selection and then move your cursor (and expand selection) using navigation keys.

But let's say you started new Vim session, and you used `3v`. Now when you move somewhere else, and type `1v`, you would expect only 1 character to be selected. But, what will happen is that 3 characters will be selected (like in previous command). Also, if you'd now run `2v` command, it would select 6 (2 x 3) characters instead of 2. For more information, take a look at `:help v`.

Selecting lines

We'll cover two ways of selecting lines.

If you'd like to select the whole line, assuming your cursor is at the beginning of it, you could press `v$` to select all characters to the end of the line.

However, a better way is to use `V`. Just by hitting `V`, you are able to select the entire line, while your cursor can be positioned anywhere on the line.

Although these two commands seem to do the same thing, there's a difference. The best way to understand it is to actually try it. For example, copy one line using `v$y` and paste it. Then do the same using `Vy` and paste it.

You'll notice that characters selection (`v$y`) is pasted differently than the line selection (`Vy`).

Visual block selection

Great Vim feature is Visual mode blockwise. Hitting *Ctrl-v* will enable you to make selection of your content in a block. The best way to understand it is to try it. And you actually did, in the example we covered in *Your first Vim session* chapter.

For example, there are many commands you can apply to a block selection. One of them is **c**hange. I suggest you go through the example from *Your first Vim session* once more now.

Extend a current visual selection

For instance, you have made a selection from line 10 to line 20. But you realized that you want to expand the selection from line 8 to line 22.

The visual selection can only be expanded with the cursor and the cursor is only on one side. But you can move the cursor to the other side of the selection by pressing `o`.

I assume that you selected lines from line 10 to line 20. As your cursor is on line 20, you can go down to line 22, and select additional lines. Now, press `o`, and your cursor will move to the top of selection, at line 10. Now you can move your cursor up two lines, and select lines 9 and 8 as well.

Run commands across selection

You can run commands across visual selection just like you're running them in Normal mode. Let's say we have the following lines written in Perl and we forgot to append `;` to each of them:

```
my $a
my $b
my $c
```

We can visually select all 3 lines using *Shift-v*, and then run command:

```
:normal A;
```

This will execute a command `A;` (append `;` to the end of a line) for each line.

If you'd like to run the same command over the entire file content, you could run:

```
:%normal A;
```

The dot command in Visual mode

As we already experienced the power of the dot command, it's also very handy to use it in Visual mode.

In chapter *Do You Speak Vim*, we had an example where we wanted to append `;` to three lines of code:

```
my $i
my $learn
my $quickly
```

In order to append the `;` to the first line, we can run the command:

```
A;<Esc>
```

Now, we can visually select second and third line (using *Shift-v* and arrow keys, or `j` and `k`), and run:

```
:normal .
```

to run the dot command over visually selected lines.

Even better, to make this operation faster, you could add this to your `.vimrc`:

```
" make . to work with visually selected lines
vnoremap . :normal.<CR>
```

With this mapping, you tell Vim: When I press . in Visual mode, execute it as I would run it in Normal mode, over the selected lines.

You can learn more about `:normal` command in chapter *Effective multiple file editing*.

Move visual selection

You made a selection, and you want to move it by a couple of lines up or down, instead of copy-paste dance. What do you do? Here's a neat solution, add this to your `.vimrc`:

```
" Move visual selection
vnoremap J :m '>+1<CR>gv=gv
vnoremap K :m '<-2<CR>gv=gv
```

Next time you need this, after you select desired lines, hit `J` to move them one line down, or `K` to move them one line up.

Visual mode possibilities

There are many possibilities offered by Visual mode which we didn't cover. Depending on your workflow, you might use it more or less often, and in different ways.

After understanding everything in this chapter, a good next step is to actually check which commands, objects and operators you can use in Visual mode, by checking out `:help visual-operators`.

Chapter 16

Mappings

Sooner or later, you'll get an idea to set up your new shortcuts, or replace the existing Vim shortcuts. To do this, you need to know the basics of mapping. To create a shortcut in Vim world is analogous to creating a mapping. To define our own mappings we'll use different types of `:map` command.

Example 1: Use `nmap` command to define mapping for Normal mode only

Task: Show Vim version info when `v` is pressed.

Info: Info about Vim version we can get with `:version` command.

1. Open new Vim session.
2. Execute this command `:nmap v :version<cr>`. (`n` in `nmap` stands for Normal mode).
3. Now, press `v`. You'll see the info about Vim version, as the `v` mapping will execute `:version` command.
4. Remove this mapping, by running this command: `nunmap v`.

Summary: This is a simple example for a mapping. By default, `v` command starts Visual mode. Once we set our mapping, there is no way to enter Visual mode anymore. Be cautious when changing the Vim default mappings. Of course, if there's a Vim command you don't use, it's usually a good choice for remapping it to your own choice.

In step #4 you can see how we removed a mapping. That's the purpose of command `nunmap`. To remove a mapping for `v`, we used command `nunmap v`. This way we removed the mapping where pressing `v` shows Vim version info and restored the default behavior (entering Visual mode).

There's just one more very important thing you need to know about the `nmap` command. It's recursive. This can be confusing for beginners, so let's make it clear through another example.

Example 2: Non-recursive mapping

Task: Create a mapping for `w` to echo `Hello world!`. Create another mapping for `a` to replace `w` default behavior.

Info: Default behavior for `w` key is to move cursor one word forward.

1. Open new Vim session.
2. Execute this command `:nmap w :echo "Hello world!"<cr>` to create the first mapping.
3. Execute command: `nmap a w` to map default behavior of `w` key to `a` key.
4. Insert a couple of words in your current session, so you can test if `a` will take a default behavior of `w` and move the cursor one word forward.
5. Now, in Normal mode, press `w`. You'll see `Hello world!` in the status line. Good, that works.
6. Then, press `a`. You'll notice that cursor doesn't move, but `Hello world!` is echoed again in the status line.

Summary: As we mapped `w` to another action already, and we know that `nmap` command is recursive—it means that whichever new mapping we create for `w`, it will always execute the existing mapping for `w` and not it's default behavior.

The proper way to resolve this problem is to use a *non-recursive* mapping command instead.

So now, use the `nunmap a` command to remove the mapping created in step #3. Then, create a new mapping using: `nnoremap a w`. Back to Normal mode, now when you hit `a` key, a default behavior of `w` key will be applied and your cursor will move one word forward.

Here's the table of both recursive and non-recursive commands you can use for mapping, as well as for unmapping:

Recursive	Non-recursive	Unmap	Modes
`:map`	`:noremap`	`:unmap`	normal, visual, operator-pending
`:nmap`	`:nnoremap`	`:nunmap`	normal
`:xmap`	`:xnoremap`	`:xunmap`	visual
`:imap`	`:inoremap`	`:iunmap`	insert
`:cmap`	`:cnoremap`	`:cunmap`	command-line
`:omap`	`:onoremap`	`:ounmap`	operator-pending

I recommend that you always use non-recursive mappings, except if you actually need recursion.

To make your mappings permanent, instead of typing and executing the mapping commands, just put them in your `.vimrc` file.

If you want to preview your current mappings for Normal mode, just run `:nmap` command.

For example, running a command like:

```
:nmap <leader>
```

will show you all normal mappings that start with the mapped leader key.

In order to disable a standard mapping, you need to map it to the special `<nop>` character like: `:noremap <left> <nop>`. This command disables left arrow key.

For further information on this topic, you could check `:h mapping` and `:h key-notation`.

Chapter 17

Folding

Collapsing multiple lines of text into a single line is called folding. This feature can be very handy, when you want to “hide” parts of your text or code which are not important for you. This way you can get a better overview of the structure of your text.

When you make folds, the text is still there in the buffer, unchanged, but just not visible, until you open your fold.

The hardest thing to remember when it comes to folds is that every related command starts with `z`. But here’s a hint: `z` looks like a *folded* paper, looking from the side :)

You’ll agree, that’s not so hard to remember.

Before we get into all the details and options, first give it a try:

1. Open any existing file where you have at least one paragraph of text.
2. Place your cursor anywhere in that paragraph.
3. While in Normal mode, type: `zfap`.

You’ll see that the paragraph got replaced by a highlighted line. You have just created a fold!

What this command is exactly and how it works, we’ll cover in the following paragraphs.

Here are a couple of configuration lines to get you started, put them in your `.vimrc`:

```
set foldenable        "Enable folding
set foldlevelstart=10 "Open most of the folds by default. If set to 0, all folds will be closed.
set foldnestmax=10    "Folds can be nested. Setting a max value protects you from too many folds.
set foldmethod=manual "Defines the type of folding.
```

As you can see, the last line of configuration from above is the `foldmethod` option.

This option defines folding for the current window. Here are the possible values you can set:

Value	Description
`manual`	folds are defined manually using commands
`indent`	groups of lines with the same indent level form a fold
`syntax`	folds are defined by syntax highlighting
`marker`	special characters can be added to text, to mark start/end of folds
`expr`	folds are defined by a user-defined expression
`diff`	used to fold unchanged text when viewing differences

We're going to cover only the three most important options here.

Manual folding

Making folds manually is usually a good way to go, if you don't need folds very often, and you don't want automatic folding.

If you set the option `foldmethod` to `manual`, then, in Normal mode, you can create a fold by typing: `zf{motion}`, where *{motion}* represents the selection of lines you want to fold.

For example, a command:

```
zf5j
```

will create a fold of current line together with five following lines.

Or, if you have a code where curly braces `{ }` are used to delimit code blocks, then you could create fold of that code block by:

1. Placing your cursor on any line, anywhere inside the code block
2. Running command `zfa{` or `zfa}` (both will work).

Let's deconstruct this command:

- `z` - I assume you remember about why `z` is used.
- `f` - comes from **f**old.
- `a` - comes from **a**round (see Modifiers in *Do you speak Vim* chapter)
- `{` or `}` - is the character which surrounds the text we want to fold.

As we already said, if you would like to make a fold of a paragraph, you would then run:

```
zfap
```

Now you understand that `zf` is an operator used to create folds, while `a` stands for **a**round, and `p` stands for **p**aragraph. Using the command from above, we basically tell Vim to "create a fold around paragraph".

At this point you might wonder, but what if there's a closed fold and you want to edit the folded paragraph?

Vim is smart enough: you don't need to manually open the fold to start with editing. You can position your cursor on the fold where you want to make an edit, and enter Insert mode—the fold will open automatically.

Here are the most important folding commands you can run in Normal mode:

Command	Description
`zo`	**O**pen current fold under the cursor.
`zc`	**C**lose current fold under the cursor.
`za`	Toggle current fold under the cursor.
`zd`	**D**elete fold under the cursor. (only the fold, text is unchanged.)
`zj`	Move the cursor to the next fold.
`zk`	Move the cursor to the previous fold.
`zR`	Open all folds in a current buffer. (**R**educe all folds)
`zM`	Close all open folds in a current buffer. (Close more and **M**ore folds)
`zE`	Delete all folds the current buffer
`:fold`	In Visual mode: fold selected lines.

Folding by indentation

If you need a lot of folds, then creating them manually can be a lot of work. And our goal is efficiency. So, if that's the case, you might need to use indent folding.

If your code is properly structured, and you use indentation, this is a great method for you. It will create folds for every sequence of lines with the same indent. Lines with a bigger indent will become nested folds. This works pretty good with many programming languages.

You would need to set the option `foldmethod` to `indent`, so lines with the same level of indentation can be folded together.

For example, the following lines:

```
This is line one
  This is line two
  This is line three
  This is line four
This is line five
```

would be folded as:

```
Line one
+ Line two, three and four
Line five
```

Sometimes it's hard to see or remember where a fold is located. To make this easier, you can enable this command:

```
:set foldcolumn=2
```

It will show a small column on the left side of the window, beside line numbers, to visually indicate folds. A `+` is shown for a closed fold, while `-` is shown at the start of each open fold. All lines of the fold will be marked with `|`. It's a good idea to set `foldcolumn` to at least the level of folds you want displayed. If you have four levels of folds, then set it to `4`.

Syntax folding

Vim uses a different syntax file for each programming language, where it defines the colors for various items in the file. So, based on your language, Vim will fold your code automatically. Of course, you can configure some of the options.

To start with syntax folding, instead of the configuration from the beginning of this chapter, you can add these lines to your `.vimrc`:

```
set foldmethod=syntax
set foldlevelstart=1

let perl_fold=1               " Perl
let perl_fold_blocks = 1      " Fold blocks in if statements, etc. in Perl
let sh_fold_enabled=1         " sh - enable function folding
let vimsyn_folding='af'       " Vim script
let r_syntax_folding=1        " R
let ruby_fold=1               " Ruby
let php_folding=1             " PHP
let javaScript_fold=1         " JavaScript
let xml_syntax_folding=1      " XML
```

This is only an example. Remove or add the options for languages you actually use. For more information on which languages and options you can use, check `:h syn-file-remarks`.

Persistent folds

Folds are not persistent by default. Once you close your Vim session, they will be gone. When you reopen the file where you were making folds, they'll be gone.

If you'd like to keep your folds persistent through Vim sessions, once you create them, you can run command: `mkview`. This will save your folds from the current buffer to your `viewdir` (see `:h viewdir`).

Next time you open file with folds you saved, use the command `:loadview` to reload them.

You can go a step further, and automate this process. All you need to do is to add this to your `.vimrc` file:

```
augroup auto_save_folds
autocmd!
autocmd BufWinLeave * mkview
autocmd BufWinEnter * silent loadview
```

This will load the existing folds (if any) when you open a file.

> *Related tip:* Using the **:mkview** command, you can store up to ten views on one file. For example, to save the current setup as the second view run **:mkview 2**. Or, if you want to load the first view, you could run **:loadview 1**

For more info check help on `viewoptions` and `viewdir` options.

Chapter 18

Effective multiple file editing

This chapter will cover a couple of commands which are very useful when you're editing multiple files at once. For each type of list in Vim, there's an appropriate command which gives us the possibility to execute commands in bulk. Here they are:

- `:argdo` - for argument list
- `:bufdo` - for buffer list
- `:windo` - for window list

We will also cover two more commands which you'll use often when editing multiple files:

- `:norm[al]` - for running commands in Normal mode
- `:exe[cute]` - for executing commands

Pay attention here, and make sure to understand the difference between `:argdo` and `:bufdo`, because this is unclear even to some advanced Vim users.

The *execute* and *normal* commands

The execute command

The `exe[cute]` command is used to evaluate a string as if it's a Vim command. You could run a command like this:

```
:execute "echom 'Hello world!'"
```

to echo the string `Hello world`. In this example, we used internal Vim command for echoing content `echom`.

This command is a very handy tool, because it lets you create commands out of an arbitrary string.

The normal command

If you try running the command:

```
:normal gg
```

you'll see that your cursor will jump to the first line in your current buffer. As you might guess already, `norm[al]` command simply takes a sequence of keys you've typed and treats them as a Vim command you would enter in Normal mode.

Of course, this command will pick up your personal mappings if they exist. That brings us to the next question: what happens if a user has remapped `gg` command to delete one line, instead of jumping to first line of the buffer?

In that case, the example from above will execute the command for deleting the line, or whatever else was specified for the mapping.

But, even if there were a mapping for `gg` command, which is different than default action of jumping to the first line, you can still use the default mapping of the command. Here's how:

```
:normal! gg
```

This way, Vim will move your cursor to the first line of a buffer, even if there's already a different mapping for `gg` sequence of keys. So if you use `!`, mappings will not be used.

You can also use ranges to execute Normal mode commands for each line in the specified range. For more details on this command, take a look at `:help normal`.

argdo vs bufdo

As you already learned, Vim has different types of lists for different purposes. Another very important list is the so called "arguments list." This list holds the files which you specify when you start Vim. We'll call it **arglist** from now on.

For example, if you start Vim like this:

```
$ vim my_file1.txt my_file2.txt
```

Vim will add to arglist two items (`my_file1.txt`, `my_file2.txt`). Once you're in Vim, you could show the contents of arglist by running `:args` command.

Now, when you open these two files, Vim creates two buffers, for each of the files. As we mentioned before, you can show the buffers list by running `:buffers` command.

At this point, both lists, for arguments and buffers, contain items `my_file1.txt` and `my_file2.txt`.

If we additionally, from Vim, open another file like:

```
:e my_file3.txt
```

Vim will create one more buffer for this file. It will also add another item in buffers list. So now, the buffers list will contain three items (for all three files we've opened), while arglist will still contain two items (with `my_file1.txt` and `my_file2.txt`).

And here we finally come to the point:

- `argdo` command affects only the files present in arglist.
- `bufdo` command affects only the files present in buffers list.

In our example, editing files via `argdo` command would affect only `my_file1.txt` and `my_file2.txt`, while `bufdo` command would affect these two, plus `my_file3.txt`.

Now let's take a look as some useful examples of these commands.

bufdo examples

When you open a file in Vim, you've actually created a Vim buffer, as we already said. For each new file you open, its buffer is added to the internal buffers list.

Let's say you have opened multiple files, so you have multiple buffers. At one point, you want to execute a command which will affect all the active buffers (in buffers list), you need to use `:bufdo` command.

For example, it's the end of your work day, you want to save your work and go home. You've already edited multiple buffers, but you didn't save your changes. Of course you shouldn't switch from buffer to buffer and run `:wq`. In order to save changes in all your active buffers and exit Vim, simply run:

```
:bufdo wq
```

This way, we've just told Vim to execute `:wq` command in each active buffer. Alternatively, you could also run the command:

```
:wqa
```

which will **w**rite changes in **a**ll buffers and **q**uit Vim.

Paste to the end of each buffer

```
:bufdo exe ":normal Gp" | update
```

Let's break it down:

- `:bufdo` - execute commands over all active buffers
- `exe` - the "execute" command, which will execute the command between double quotes.
- `":normal Gp"` - this will be executed by `exe` command. We use `:normal Gp` to tell Vim: run `G` (jump to the end of a file) and `p` (paste) in Normal mode.
- `| update` - when the previous commands are executed, we use `|` to execute another command (`update`), which actually writes the changes in the buffers. You could also use `w` instead.

> *Related tip:* Whenever you execute **:w**, Vim will actually write the buffer to the file, no matter whether the buffer was changed from the last saved state. This means it will update the timestamp of the file to the time when you run **:w**, even if the contents of the file did not actually change.
>
> On the other hand, when you execute **:up[date]**, the Vim will update the file timestamp ONLY if the file has been changed.

Executing macro over all active buffers

```
:bufdo execute "normal! @a" | w
```

As you can see, the structure of the command is the same as in the previous example. This time we run `@a` command in Normal mode to execute the macro recorded in register `a`. Also, this time we use `w` instead of `update` (for no specific reason).

We have already covered how to manage buffers in chapter Buffers, so we won't repeat the commands for adding and deleting buffers.

argdo examples

All of the examples for `:bufdo` command can be applied to files stored in arglist, by replacing `:bufdo` with `:argdo` command.

Before we get to some examples, it's important to mention that we can, just like with buffers list, add and delete items in arglist.

Here are a few useful commands:

Command	Description
`:args`	Show files in your current arglist
`:args /path/to/files/*/*`	Replace old arglist files with new ones
`:arga[dd] /path/to/file.txt`	Add `file.txt` to your arglist
`:argd[elete] /path/to/file.txt`	Remove files from your arglist
`:argdo update`	Save changes to all arglist files
`:argdo undo`	Undo last operation in each arglist file

Here's an example: You're working on a project, and have multiple files in your `git` branch. You want to replace string `Bad` with string `Good` across the entire project.

We can do this in two steps, here's the first one:

```
:args `git grep -l Bad`
```

This command runs a shell command `git grep -l Bad`, which will provide us the list of files containing string `Bad`, and place those files in argslist.

Then, we can run:

```
:argdo %s/Bad/Good/gc | update
```

and perform the substitution across all files in arglist, and save the changes.

But, what if you add multiple files to the arglist, and you want to replace `bad` with `good` in all of them where `bad` appears, without using external shell commands? If you would run a command like this:

```
:argdo %s/bad/good/g | update
```

Vim will say there's an error when the search operation fails for a file which doesn't contain the string `bad`. This will prevent completing the substitution over all files in the arglist. A better way to run this command would be:

```
:argdo %s/bad/good/ge | update
```

We have included `e` flag, which tells Vim to not issue an error if the search for the pattern fails. If you'd like to perform substitution selectively, and skip some of the matches, you could just add the `c[onfirm]` flag.

One last example will show you how you can use shell commands from Vim.

```
:argdo exe '%!sort' | w
```

This command will sort the contents of all files in your arglist. It calls the shell command `sort`.

windo command

Let’s keep it short.

The command `:windo` is similar to `:argdo` and `:bufdo`, with one important difference: it only affects the visible buffers. As we already mentioned, Vim can have many buffers open, but if there’s only one window, then only one buffer is shown at the time.

So, if you have a Vim session with multiple windows open, then the commands you run with `:windo` will be applied only to those buffers which are currently visible in your windows.

Let’s summarize. There’s so much power in these commands. We’ve covered the most important concepts and some real word examples. Take time to understand and learn how you can use these commands, and very quickly you’ll become very effective at editing multiple files.

Chapter 19

Productivity tips

Here we'll go through different tips which can give a boost to your productivity. If you find some tip you don't like or it doesn't feel right in your workflow—just don't use it. These tips will improve your productivity and efficiency with text editing, but only if they'll fit your mindset and the way you work.

Read them all, give them all a try for a few days. And if they stick—great! If they don't, just forget about them for a while—and try them again in a few months.

Relative numbers

Most of the text editors use absolute line numbers. Vim doesn't, by default. But, what if relative numbers are better? For many advanced users, relative numbers are the correct way of using Vim.

In Vim, many commands can be prefixed by a number. Move 5 lines below the current line, use `5j`. Delete the next 5 lines, use the command `d5j`. Move 8 lines above the current line, use `8k`. Indent next five lines, use `5>>`. You can also use `+` to refer to lines below your current line, and `-` to refer to the lines above your current line.

You get the point. With relative numbers, you simply see how far the other lines are. If you would use absolute numbers, you would have to mentally count the number of lines you want to manipulate. Using relative numbers will save you from counting the number of lines above or below your current line.

To start using relative numbers, put the following line to your `.vimrc`:

```
set number
set relativenumber
```

This will work since Vim 7.4—you can enable both relative and absolute numbers. This way Vim shows the absolute number for the current line, and relative numbers for other lines.

You might want to go a step further. An even better way might be to enable relative numbers only in Normal mode, and absolute numbers only in Insert mode. Here's what you'd need to put in your `.vimrc`:

```
augroup toggle_relative_number
autocmd InsertEnter * :setlocal norelativenumber
autocmd InsertLeave * :setlocal relativenumber
```

That's my preferred way of using line numbers. Give it a try for a day or two, and see what works better for you.

Using the Leader Key

Leader key gives you a namespace for custom Vim mappings. There are no default mappings which use leader key, so you can map your shortcuts without need to worry about shortcut conflicts.

Defining a leader key is one of the best things you can do to boost your productivity in Vim. You can define leader shortcuts for your most used commands.

The default leader key is a backslash `\`, but it's a good practice to remap it to either `Space` or `,` key. It mostly depends on your personal preferences. To activate a defined shortcut, you basically need to press `Leader key` and then a specific mapping key.

For example, instead of executing `:w` to save file, you could map this command and execute it like `Leader w`. In order to configure a mapping like this, you'll first need to map your leader key. In this example, let's make *Space* your leader key. Just add this line to your `.vimrc`:

```
let mapleader = "\<Space>"
```

Then, to enable the shortcut from the example above, all you need is to add this simple mapping to your `.vimrc`:

```
nnoremap <Leader>w :w<CR>
```

If you're a beginner, I'd suggest that you use *Space* as your leader key, for two reasons:

- *Space* doesn't have any big use in Normal mode by default
- It's symmetrical and equally easily reachable for both hands

Now, depending on your workflow, and commands you use the most often, you should define your shortcuts over time. This will drastically improve your efficiency.

Just to give you an idea, here are some shortcuts to make working with system clipboard easier:

```
vmap <Leader>y "+y
vmap <Leader>d "+d
nmap <Leader>p "+p
nmap <Leader>P "+P
vmap <Leader>p "+p
vmap <Leader>P "+P
```

If you want to know even more about the leader key, see `:help mapleader` and `:help maplocalleader`.

Automatic Completion

There are multiple ways (as usual) to auto-complete the words you're typing in Vim. We'll cover three different ways of using the auto-complete feature:

- using known words
- using a dictionary file
- using a thesaurus

No matter what are you writing, even after a small amount of text, there will be need for using some repeated words. Auto-completion using known words is the simplest to use—all you need to do is to press *Ctrl-n*. You simply need to type a first few letters of a word (which is already present somewhere in your earlier text) and press *Ctrl-n*.

Let's see an example. First type a sentence:

```
I love to learn new things.
```

Then, write:

```
Today I le
```

press *Ctrl-n* and then type Vim.

Vim will auto-complete the word you're typing because it already "knows" the word *learn*. So you'll get the sentence:

```
Today I learn Vim.
```

Pressing *Ctrl-n* tells Vim to search for a matching word forward through the file. If you have a very big file, and you know that you've recently typed a word which you want to auto-complete, then instead of *Ctrl-n* use *Ctrl-p*. Pressing *Ctrl-p* will also auto-complete your word, but it will look for known words backwards from your position.

Also, repeated *Ctrl-x Ctrl-n* is very useful. It only works after previous completion, usually *Ctrl-n*. You can use it for partial line completion when beginning is different or when you want a different ending.

Another way to use auto-complete is using a dictionary. You need to find a dictionary with words in your language or in the topic you're writing about.

Once you have the dictionary file, all you need to do is: `set dictionary+=/path/to/dictionary/file`

After running this command, Vim suddenly knows a lot of words which can be auto-completed. But, in this case, everytime you want to auto-complete the word, you need to press *Ctrl-x Ctrl-k*.

Pressing *Ctrl-x* gets you into "Completion mode", while pressing *Ctrl-k* tells Vim to auto-complete a keyword in the dictionary.

Vim offers a couple of more auto-complete options. After you press *Ctrl-x*, beside *Ctrl-k* we've already mentioned, you can also use:

- *Ctrl-l* to complete whole lines of text
- *Ctrl-i* to complete words from current and included files
- *Ctrl-t* to complete words from a thesaurus
- *Ctrl-f* to complete the name of files the current user has access to

The last auto-complete feature I want to mention is thesaurus auto-completing. This feature enables you to define a list of synonyms, which can be later used for auto-complete.

1. First you need to define a thesaurus file. You need to write a list on synonyms separated by commas or spaces in a single line.
2. Specify thesaurus file location in your `.vimrc`:

```
set thesaurus+=/home/jole/my_thesaurus.txt
```

3. Use thesaurus as you type, by pressing *Ctrl-x Ctrl-t* to replace the word with its synonym.

Of course, you can download ready thesaurus files from the Internet. This feature can be very handy for software developers. For example, you could define all the existing keywords and functions for your favorite programming language, and speed up your typing with auto-complete.

Using File Templates

Every time you start working on a new project or a new file, you'll most probably add some headers and different kinds of stuff, depending on the programming language.

In order to speed things up, you can set up template files, based on the file type. For example, everytime I create a new Perl file, I preload from template something like:

```
!/usr/bin/env perl
#===============================================================================
#         FILE: filename.pl
#        USAGE: ./filename.pl
#  DESCRIPTION:
```

```
#      OPTIONS: ---
# REQUIREMENTS: ---
#         BUGS: ---
#        NOTES: ---
#       AUTHOR: YOUR NAME (),
# ORGANIZATION:
#      VERSION: 1.0
#      CREATED: 04/18/2017 08:58:16 PM
#     REVISION: ---
#===============================================================================
use strict;
use warnings;
use utf8;
```

Here's how to enable this feature:

1. Create a directory in your `.vim` directory, where you will store your files. For example:

```
mkdir ~/.vim/templates
```

2. Create the contents of the file which you want to be preloaded. In this example, let's make a simple template for .html files:

```
<html>
    <head>
        <title></title>
        <meta name="generator" content="Vim" />
        <meta name="author" content="Jovica␣Ilic"/>
    </head>
    <body>
        <p>Your content here.</p>
    </body>
</html>
```

name it `html.tpl` and save it in the `templates` directory you've just created.

3. Tell Vim to autoload this template every time you create a new `.html` file, by adding the following line to your `.vimrc`:

```
:autocmd BufNewFile *.html 0r ~/.vim/templates/html.tpl
```

And that's all. If you want to add another template for different file types, just repeat the last two steps.

Repeat the last Ex command

As you already know, Ex commands are those which you execute in Command mode, like `:sort`. Sometimes you'll need to repeat the last Ex command multiple times. Instead of typing it again, you can repeat the last Ex command by typing `@:` in Normal mode.

This is possible because of the read-only register `":`, known as the colon register. We've only mentioned it in the *Registers* chapter, so we'll cover it here.

Whenever you execute a command in Ex mode, such as `:write`, it populates the colon register. Let's say you've enabled relative line numbers with `set relativenumber` command in your current Vim session. Now you want to make this change persistent and put it in your `.vimrc` file. Once you open `.vimrc` file, you can just run `":p` to paste the contents of the colon register.

Paste text while in Insert mode

You're in Insert mode and you want to paste yanked text without moving to Normal mode. You can do that with *Ctrl-r* `0`. If yanked text contains new line characters, *Ctrl-r Ctrl-p* `0` will fix the indentation issues.

Delete in Insert mode

Sometimes it comes in handy to delete some characters as we type. That's why it's good to know that you can delete also while you're in Insert mode. Here are the most useful mappings:

- *Ctrl-h* - delete back one character (just like *Backspace*)
- *Ctrl-w* - delete back one word
- *Ctrl-u* - delete back to the start of line or the start of current insert

Repeatable operations on search matches

Let's say you're searching for a string which you need to edit at multiple locations through the file:

```
/MyString
```

To edit the string, you will usually use a repeatable operation, like `cw` or any other which helps you achieve your change. Now, to repeat that change for the next match, you'll first have to jump to the next search result using `n` then press the `.` (the dot command) to repeat the change.

But in this case, there's a better way: Instead of `cw` use `cgn`. Then just press the `.` and Vim will jump to the next match and change it for you. While `c` stands for change, `gn` searches forward for the last used search pattern and starts Visual mode to select the match.

This also works if you (instead of searching with `/`) use `*`, which searches for the word currently under the cursor. For more info, check `:help gn`.

Copy lines without cursor movement

Your cursor is on line 10. You want to paste line 20 to one line below your current cursor position.

Here's how to do that using `co[py]` command:

```
:20co.
```

You can also use ranges. Let's say your cursor is on line 10. You want to paste text from line 20 to 25 **under** line 10. Here's how:

```
:20,25co10
```

It gets even better: `:t` is an alias of the `co[py]` command, so you could save some keystrokes. You could run the commands from above and achieve the same result if you'd replace `co` with `t` in them.

This command works with relative line numbers as well. For example, to paste the line, which is 10 lines above your current line, to a line below your current position:

```
:-10t.
```

One last example: if you're at line 45, `:35,t.` will make a duplicate of lines 35 to your cursor (that is, from 35 to 45 inclusive) and put it after your current cursor.

So, imagine this case: you have a function and your cursor is one line below it. You also have relative line numbers enabled. You see that the function starts 15 lines above your current line. To make a copy of the entire function and place it after your current line, you could run

```
:-15,t.
```

Move lines without cursor movement

The usage of command `m[ove]` is similar to `co[py]` command. For example, to move a line 6 to line 28, you'd run:

```
:6m28
```

It also supports ranges and relative lines. Here's an example using both:

```
:-10,-5m+7
```

This command would take five lines which are located between lines 10 and 5 above your current position, and move them to 7th line under your current position. After this command, the position of your cursor will change. In order to come back to the original location where your cursor was before running this command, simply press `''`.

Delete lines without cursor movement

Wouldn't it be cool to be able to delete lines without moving your cursor?

Similar to `:copy` and `:move` commands, you can also run `:delete` command to delete lines without jumping to those lines.

For example, to delete lines 5 to 10, run:

```
5,10d
```

However, this command would leave the cursor at the deleted line location, so you'd need to use `''` to jump back to the previous position.

Vim write through

Vim lets you write through existing text, so you don't have to delete it first. All you need to do is to press `R` while in Normal mode, and start typing. This will actually put you in Replace mode. Once you're done, just press `Esc`.

Run the same command on multiple lines

Using the `norm` command, you can run the same command on multiple lines at once. Let's say you want to delete all strings inside the single quotes `'` on all the lines. Here's what you'd need to do:

```
:%norm di'
```

- `%` - defines range: all lines.
- `norm[al]` - command which tells Vim to repeat the following command in Normal mode
- `di'` - delete inside single quotes `'`

Generating numbered lists

This tip is one of those which you probably won't use often, but you might need it from time to time. That's why it's important you're aware that Vim can do even something like this. We'll see different methods showing how to insert a list of increasing numbers.

For example, to insert a list of ascending numbers, you can run a command like this:

```
:put =range(1,10)
```

You can also insert numbers after a particular line number, for example command:

```
:28put =range(6,87)
```

inserts list of numbers from 6 to 87 after line number 28.

Generating IP address list

You could also use loop to generate different kinds of lists. In this example, we want to generate a list of IP addresses, starting from `192.168.0.1` to `192.168.0.100`:

```
:for i in range(1,100) | put ='192.168.0.'.i | endfor
```

The result will be list of IP addresses in the mentioned range, with one IP address per line.

Increasing or decreasing numbers

It's one of those features you'll use regularly. In Normal mode, hitting *Ctrl-a* will increment the next number. Hitting *Ctrl-x* will decrement the next number. In order for this to work, the cursor can be at the number, or to the left of the number, on the same line.

These keys work with a count. For example, pressing `4` then *Ctrl-a* will increment the following number four times (add 4).

Why Vim 8 is great

Vim 8 has introduced many new features, which are out of the scope of this book. But this one feature I find very useful.

Let's say you have something like this in your code:

```
array[0] = 0;
array[0] = 0;
array[0] = 0;
array[0] = 0;
array[0] = 0;
```

and you want to get something like this:

```
array[1] = 0;
array[2] = 0;
array[3] = 0;
array[4] = 0;
array[5] = 0;
```

Best way to do it:

1. Place your cursor at the first line, at `0` for which you want to become `1`.
2. Press *Ctrl-v* to enter Visual block mode, move the cursors down to select the rest of the zeroes, to the last line.
3. Now, press `g` and then press *Ctrl-a* (the shortcut for increasing numbers).

Faster delete/change to the end of the line

You could delete the entire line from your cursor position by running command `d$`. That takes three keystrokes: `d`, *Shift*, `$`. A faster way do to it with two keystrokes: `D`. Same goes for **c**hange command. You can change (delete text and enter Insert mode) from cursor position to the end of the line by hitting `c$`. Faster way: `C`.

Repeating characters

You want to add 8 new lines under the current line? Don't even think of hitting *Enter* 8 times. Instead, run:

```
8i<Enter><Esc>
```

You need to insert 20 "-" (dash) characters? Please, don't hit `-` twenty times. Run:

```
20i-<Esc>
```

These examples just showed you the possibilities. Next time you need to insert more than a few identical characters, remember these.

Clear highlighted searches

After you've found what you've searched for, you'd soon want to remove the highlighting on the previously searched word. The easiest way to do it is to search for a string which doesn't exists, like:

```
/adasdada
```

This string won't be found and highlighted, and previously highlighted string will not be highlighted anymore. Although this is the easiest, it's not really a cool way of doing it. A much

nicer solution could be that you create your own shortcut (in Vim world we call this mapping) to run a command which will clear previously highlighted strings.

For example, you could add this to your `.vimrc`:

```
nmap <silent> ,/ :nohlsearch<CR>
```

Next time, you can quickly type `,/` to clear the highlights. Calling `nohlsearch` command like this doesn't change the option value, so as soon as you use a search command again, the highlighting comes back.

Execute multiple commands at once

You have a possibility to execute more than just one command. All you need to do is to use `|` between the commands. For example, let's say you have a sentence like *"Atom is bad and slow."* and you want to run three different substitution commands.

```
%s/Atom/Vim/c | %s/bad/good/c | %s/slow/fast/c
```

This example substitutes `Atom` with `Vim`, then moves on to replacing `bad` with `good`, and finally it goes to replacing `slow` with `fast`. Notice that the second command (and subsequent commands) are only executed if the prior command succeeds.

This won't work for commands like `:argdo` or `:bufdo`, as they use `|` to execute a series of commands.

However, you can use `:execute` command:

```
:exe "argdo %s/bad/good/e" | exe "bufdo %s/old/new/e" | echo "done"
```

to perform multiple commands at once, even using `:argdo`, `:bufdo` and similar commands.

External program integration

Vim has a great integration with system CLI (command-line interface) programs, which you can use to modify the current buffer. Here's how to do it:

```
:%!<command>
```

Here are few examples to give you an idea on how powerful this feature can be:

- `:%!sort -k3` - sort the buffer based on column 3
- `:%!column -t` - format the text in columns (useful when working with tabular data)
- `:%!ls|grep .txt` - insert the list of txt files from the current directory

With a bit of creativity, you can do wonders!

Auto remove trailing whitespace

No programmers like to have trailing whitespaces in their code. You can set Vim to highlight trailing spaces, and also to remove them when you perform a save. All you need is to add two lines to your `.vimrc`:

```
" highlight trailing whitespace
match ErrorMsg '\s\+$'
" remove trailing whitespaces automatically
autocmd BufWritePre * :%s/\s\+$//e
```

Okay, those are actually four lines, but comments are always good to have.

Open and edit archives

You need to edit a file inside an archive? Without Vim, you'd have to extract the archive, edit the file and save changes, then create a new archive with the updated file. Not anymore!

With Vim, you can open and edit files inside archive files without previously extracting them. Multiple archive types are supported, like .zip, .tar, .tar.gz, .jar, etc. For example, you could run:

```
$ vim my_archive.tar.gz
```

and you'd see all the files and directories inside `my_archive.tar.gz`. You'd be able to browse through them, open, edit and save them.

Open the last edited file

What if you'd like to open the last edited file with Vim, and jump to the latest location in that file? Well, Vim has neat shortcut for exactly this!

Simply start Vim, and press *Ctrl-o-o*.

Navigation through cursor history

As we already stated, Vim has different types of lists for different purposes. Two more lists worth mentioning are Jumplist and Changelist.

Briefly, Jumplist stores each position to which your cursor jumped, while Changelist stores every change (actually its position) which can be reverted (with undo).

You'll often get in a situation when you'd like to get back to some of your previous cursor positions. For example, you search for something, find your match in another part of the file. After you're done, you want to jump back to where you were before the search. How to perform actions like this effectively?

Easy. Learn to navigate through these two mentioned lists.

You want to move through Jumplist? Here's how:

- *Ctrl-o* - to move backwards
- *Ctrl-i* - to move forwards

To move through Changelist, use:

- *g;* - to move backwards
- *g,* - to move forwards

If you find this feature useful, take a look at `:help jumps` and `:help changes`. There are several unexpected and interesting things to discover.

Invert selection

You can easily execute the Ex commands `[cmd]` on the lines within `[range]` where `{pattern}` does NOT match using `:v` command. Here's the structure:

```
:[range]v[global]/{pattern}/[cmd]
```

For example, to delete all the lines in a file that doesn't contains the string `127.0.0.1`, you could execute the following command:

```
:v/127.0.0.1/d
```

Quickly switching buffers

It depends on your work style, but more or less often you'll be in situation to jump between two files. That's why having a shortcut to switch between the last edited buffer and your current one would be very handy.

Vim actually has the shortcut for this: *Ctrl-^*. But it's not very comfortable to use it, especially if you need to use it often.

A good idea would be to create a mapping of your preference. Here's an example:

```
"Jump back to last edited buffer
nnoremap <C-b> <C-^>
inoremap <C-b> <esc><C-^>
```

With this, you could jump between two files just by using *Ctrl-b* shortcut.

Fix indentation in entire file

To indent the current line you'd run `==`, while for indenting 4 lines below the current line you'd run `4==`. To indent a block of code, you can place your cursor at one of the braces and use command `=%`.

To fix indentation in entire file, run `gg=G`. Command `gg` will take you to the top of the file, command `=` is the indentation command, and `G` at the end tells Vim to run indentation command to the end of the file.

Chapter 20

Plugins

The world of Vim plugins is enormous. And it keeps growing. There are so many very useful plugins out there, that it's even hard to keep track of the best ones.

The first step to installing a plugin in Vim, is actually installing a plugin manager. Plugin manager is a name for the type of Vim plugins which help you install, update, and uninstall other plugins easier.

Although Vim 8 came with a native way to load third-party packages, my personal preference is to still use a plugin manager.

In this chapter we're not going to cover any Vim plugins, except one, which we'll use as an example to learn how to install a typical plugin manager for Vim.

There are many plugin managers, and few of them are really good. One of those few is called Vundle. You can get more info about Vundle at: https://github.com/VundleVim/Vundle.vim

How to install plugin manager Vundle

We assume that your main Vim directory is at `~/.vim`. First step is to run the following command in your terminal:

```
$ git clone https://github.com/VundleVim/Vundle.vim.git ~/.vim/bundle/Vundle.vim
```

Then, put the following code at the beginning of your `.vimrc` file:

```
set nocompatible              " be iMproved, required
filetype off                  " required

" set the runtime path to include Vundle and initialize
set rtp+=~/.vim/bundle/Vundle.vim
call vundle#begin()
```

```
" alternatively, pass a path where Vundle should install plugins
"call vundle#begin('~/some/path/here')

" let Vundle manage Vundle, required
Plugin 'VundleVim/Vundle.vim'


" All of your Plugins must be added before the following line
call vundle#end()            " required
filetype plugin indent on    " required
" To ignore plugin indent changes, instead use:
"filetype plugin on
```

This is the minimal example, but it shows you the most important parts. So, after you put this code in your `.vimrc`, save it and close your Vim session. Start Vim again, and run the command `:PluginInstall`. This will open a new split window and run through the installation of your new plugins, in this case only Vundle. That's all that is needed when it comes to Vundle installation.

How to install a new plugin

Most of the best plugins are hosted on github.com. One of the great plugins for advanced Vim users is called targets.vim and it's available at: https://github.com/wellle/targets.vim.

In order to install this plugin, all you'd need to do is to add a line `Plugin wellle/targets.vim` to your `.vimrc` at the right location, like:

```
call vundle#begin()

Plugin 'VundleVim/Vundle.vim'
Plugin 'wellle/targets.vim'

call vundle#end()
```

It's very important that you add all of your plugins between lines `call vundle#begin()` and `call vundle#end()`, like in the example above. Then, all you need to do is to run `:PluginInstall` command.

The most important Vundle commands are:

- `:PluginList` - List installed plugins
- `:PluginInstall` - Install plugins (append `!` to update or just `:PluginUpdate`)
- `:PluginClean` - Remove unused plugins (append `!` to auto-approve removal)

The smaller number of plugins you use, the better. That's what I think at least. Before installing a plugin to get some feature, always try to search for an existing solution in Vim. It's very often the case that you won't need a new plugin. If you want to see which plugins are the most popular among Vim users, check out https://vimawesome.com.